MobileIron AppConnect 4.6.0 for iOS SDK App Developers Guide

August 5, 2020

For complete product documentation see:
MobileIron AppConnect for iOS Product Documentation Home Page

Contents

Introducing the MobileIron AppConnect for iOS SDK

AppConnect for iOS overview

MobileIron AppConnect for iOS provides a software development kit (SDK) for securing and managing enterprise applications on mobile devices. These secure enterprise apps are called *AppConnect apps* or *secure apps*.

You can develop an AppConnect app for apps written using:

- **Objective-C**, by using the AppConnect for iOS Objective-C APIs.
- **Swift**, by using the Swift interfaces that correspond to the AppConnect for iOS Objective-C APIs. These Swift interfaces are automatically generated by Xcode when you add the AppConnect framework into your Xcode project.
- **the Xamarin development platform**, using Xamarin C# bindings of the AppConnect for iOS Objective-C APIs.
- **Cordova (or Phonegap)**, by using the AppConnect for iOS Cordova Plugin, described in the *MobileIron AppConnect for iOS Cordova Plugin Developers Guide.*

Note The Following:

- If your AppConnect app is to be distributed from the Apple App Store, due to Apple App Store requirements, your app is required to work as a regular app in addition to working as an AppConnect app.
  See Developing Third-party Dual-mode Apps.
- If your app uses an older version of the AppConnect for iOS SDK, MobileIron recommends that you always rebuild your app with the current version of the SDK. Using the current version ensures the app contains all new features, improvements, and resolved issues.
- An Apple Developer Enterprise Program account is required to distribute in-house apps. See Apple Developer Enterprise Program.

Where to get the AppConnect for iOS SDK

The AppConnect for iOS SDK ZIP file is available at help.mobileiron.com in the **Software** tab.

Check for the latest updates to this document and other developer resources on: https://developer.mobileiron.com.

The SDK is also available at https://support.mobileiron.com/support/CDL.html.

Documentation is also available at https://support.mobileiron.com/docs/appconnect/.

Legal notices are also available on https://support.mobileiron.com/copyrights/ACe.

Secure app features

Secure enterprise apps that are built using the SDK can:

- Receive app-specific configuration information from the MobileIron server.
  This capability means that device users do not have to manually enter configuration details that the app requires. By automating this process for the device users, each user has a better experience when installing and setting up apps. Also, the enterprise has fewer support calls, and the app is secured from misuse due to configuration. This feature is also useful for apps which do not want to allow the device users to provide certain configuration settings for security reasons.
- Tunnel network connections to servers behind an enterprise's firewall.
  This capability means that device users do not have to separately set up VPN access on their devices to use the app.
- Authenticate an app user to an enterprise service.
  This capability means that AppConnect app users do not have to enter login credentials to access enterprise resources.
- Handle data loss prevention.
  The MobileIron server administrator decides whether an app can copy content to the iOS pasteboard, use the document interaction feature), receive documents from other apps (Open From) use drag and drop, or print. The AppConnect library enforces the pasteboard, Open In, Open From and drag and drop policies. The app enforces the print policy.
- Control custom keyboard use by your app.
  The MobileIron server administrator can choose whether an app can use custom keyboards, and the AppConnect library enforces the choice. If the administrator does not configure this choice, your app can choose to reject custom keyboard use.
- Blur the app's screens when the app is not in the foreground.
  This blurring occurs if you specify a particular key in your Info.plist. The AppConnect library then enforces this behavior, which can be overridden by the MobileIron server administrator.
- Protect the app's data independent of device level encryption.
  You can protect your app's data using APIs provided by the AppConnect for iOS SDK. This secure file I/O capability means that data encryption for your app is not dependent on the device having a device passcode. Note that the AppConnect library and the MobileIron client app protect AppConnect-related

data, such as configurations and certificates, without any special actions by the app. The secure file I/O APIs also allow you to share encrypted data among AppConnect apps.

- Obtain derived keys for custom encryption.

  If your app uses custom cryptography, you can get derived encryption keys from the AppConnect library. This feature is useful for legacy apps that cannot easily convert to using the SDK's secure file I/O APIs. Because the keys are derived, accidental leaks have limited damage, and the keys are not weakened by brute force attacks.

- Secure sensitive data like encryption keys and passwords

  The AppConnect for iOS SDK provides APIs for heightened security of especially sensitive data. These APIs use Apple hardware capabilities (Apple's Secure Enclave) to reduce the sensitive data's attack surface, because the data is never stored in plain-text in memory.

- Log messages to the device's console and files.

  By using APIs provided by the AppConnect for iOS SDK, your app can log messages of different severity levels to the device's console. The MobileIron server administrator decides the severity levels that are written to the console, and whether the logs are also written to files.

- Provide AppTunnel diagnostics.

  By using APIs provided by the AppConnect for iOS SDK, your app can log or display diagnostic data about your app's AppTunnel configuration and usage.

- Be FIPS compliant.

  See FIPS Compliance in an AppConnect SDK App.

- Deliver derived credentials to the MobileIron client app.

  This capability is only for apps that obtain derived credentials from a derived credential provider and deliver the credentials to the MobileIron client app. Very few apps implement this capability. How to implement this capability is described in Derived Credential Handling.

AppConnect for iOS SDK advantages

With the AppConnect for iOS SDK:

- You can focus on application logic.

  The SDK handles low-level, complex work such as authentication to access AppConnect apps, certificate authentication to enterprise resources, tunneling, AppConnect passcode handling, data encryption, and getting app-specific settings and configuration from the MobileIron server.

- You use a set of simple APIs to develop a secure enterprise app.

- The app does not have to interact directly with web service interfaces to get the information it needs to behave as a secure enterprise app. Using the APIs, the app gets notified of any changes that the administrator makes on the MobileIron server to controls and configuration.

- You can create one app, with one code base, that can behave as a secure app or a regular app. This

behavior is required for secure apps that are distributed from the Apple App Store.

- For more information, see Developing Third-party Dual-mode Apps.

64-bit and 32-bit app support

Using the AppConnect for iOS SDK, you can build an app as a 64-bit app or as a 32-bit app.

MobileIron AppConnect components

The apps that you build with this SDK work with the following MobileIron components:

TABLE 1. MOBILEIRON COMPONENTS INVOLVED WITH APPCONNECT APPS

MobileIron component	Description
MobileIron Core	The MobileIron on-premise server which provides security and management for an enterprise's devices, and for the apps and data on those devices. An administrator configures the security and management features using a web portal.
MobileIron Connected Cloud	The MobileIron cloud offering that has the same functionality as MobileIron Core.
MobileIron Cloud	The MobileIron cloud offering that provides similar functionality as MobileIron Core. However, it does not support all the AppConnect features that MobileIron Core supports.
Standalone Sentry	The MobileIron server which provides secure network traffic tunneling from your app to enterprise servers.
The Mobile@Work for IOS app	A MobileIron client app that runs on an iOS device. It interacts with MobileIron Core or Connected Cloud to get current security and management information for the device. It interacts with the AppConnect library to communicate necessary information to your app.
The MobileIron Go app	A MobileIron client app that runs on an iOS device. It interacts with MobileIron Cloud to get current security and management information for the device. It interacts with the AppConnect library to communicate necessary information to your app.
The MobileIron AppStation app	A MobileIron client app that runs on an iOS device. It interacts with MobileIron Cloud. It can be used on the device instead of MobileIron Go when the MobileIron Cloud tenant supports Mobile Apps Management (MAM) but not Mobile Device Management (MDM). It interacts with the AppConnect library to communicate necessary information to your app.
The AppConnect library	The MobileIron library that your app uses to get AppConnect information. The AppConnect library is part of the AppConnect framework that your app includes. It provides your app management and security capabilities, and facilitates communication between your app and the MobileIron client app.

Note The Following:

- MobileIron Core, MobileIron Connected Cloud, and MobileIron Cloud are each also referred to as a MobileIron server.
- Mobile@Work, MobileIron Go, and MobileIron AppStation are each also referred to as a MobileIron client app.

IMPORTANT: Some AppConnect features depend on the version of MobileIron Core, MobileIron Cloud, Standalone Sentry, and the MobileIron client app.

Using a secure app

A device user can use a secure enterprise app only if:

- The device user has been authenticated through the MobileIron server.
  The user must use the MobileIron client app to register the device with the MobileIron server. Registration authenticates the device user.
- The server administrator has authorized the device user to use the app.
- The device user has entered a secure apps passcode or Touch ID/Face ID.
  The server administrator configures whether a secure apps passcode, also called the AppConnect passcode, is required, and configures its complexity rules. The administrator also configures whether using Touch ID/Face ID, if available on the device, is allowed instead of the AppConnect passcode.

The AppConnect passcode is not the same as the passcode used to unlock the device.

App responsibilities

Your app is responsible for:

- enforcing the authorization settings
- handling the data loss prevention settings
- using the app-specific configuration
- ensuring the app's data is protected by using the AppConnect secure file I/O APIs
- logging messages at the appropriate log level to protect sensitive data
- logging or displaying AppTunnel diagnostic information (optional)
- preserving and restoring the app's state when control switches from the app to the MobileIron client app and back

MobileIron client app and AppConnect library responsibilities

The MobileIron client app and the AppConnect library are responsible for:

- authenticating the user to the MobileIron server
- authenticating to enterprise services using certificates

- tunneling network connections
- AppConnect passcode and Touch ID / Face ID handling
- protecting AppConnect-related data, such as configurations and certificates
- managing the encryption key for the AppConnect secure file I/O
- controlling when sensitive log messages are written

AppConnect for iOS SDK variants

Due to Apple deprecating the UIWebView class, the AppConnect for iOS SDK is available in two variants. One with UIWebView support and another without the support for UIWebView. The AppConnect SDK without UIWebView support is available to use for apps that are submitted to the App Store.

AppConnect for iOS SDK contents

The AppConnect for iOS SDK is available as a ZIP file called AppConnectiOSSDK_V<*version*>_<*build*>.zip, where:

- <*version*> is the version number of the SDK.
- <*build*> is the build number of the SDK.

The ZIP file contains the following:

- AppConnect.framework, which you include in your app's set of frameworks.
  The AppConnect.framework includes the AppConnect library and header files.
- AppConnectExtension.framework, which you include in an extension of an AppConnect app to share files with a host app. AppConnectExtension.framework includes the AppConnect extension library and header files.
- A Documentation folder, which contains,
  - this document
    Check for updates to this document as described in Where to get the AppConnect for iOS SDK.
- A plugins folder, which contains,
  - the cordova folder, which contains the Cordova plugin, sample apps, the install_ac_cordova_plugin.sh script, and documentation
  - the xamarin folder, which contains the Xamarin C# bindings, sample apps, and C# API documentation.
    See Developing AppConnect Apps with Xamarin
- the script post_embed_actions.sh
  See Add AppConnect files and settings to your Xcode project.
- Notices.pdf, which contains SDK copyright information, software, and licenses.

- README_license.pdf, which contains the SDK license agreement.
- A Samples folder, which contains these sample apps:
  - HelloAppConnect, which demonstrates how an app uses the AppConnect framework. It displays its authorization status, its app configuration, and its data loss prevention policies.
    The sample includes both an Objective-C and a Swift version of the app.
  - DualMode example, which demonstrates the behavior of a dual-mode app.
  - SwiftFileSharing app, a Swift app demonstrating AppConnect API usage, including sharing secure files from an extension.
- The SDK_without_UIWebView folder which contains the iOS SDK variant that does not support UIWebView. The folder includes the following:
  - AppConnect.framework, which you include in your app's set of frameworks.
    The AppConnect.framework includes the AppConnect library and header files.
  - AppConnectExtension.framework, which you include in an extension of an AppConnect app to share files with a host app.
    AppConnectExtension.framework includes the AppConnect extension library and header files.
  - A plugins folder, which contains:
    - the cordova folder, which contains the Cordova plugin, sample aspps, the install_ac_cordova_plugin.sh script, and documentation
    - the xamarin folder, which contains the Xamarin C# bindings, sample apps, and C# API documentation.
      See Developing AppConnect Apps with Xamarin

Header files in AppConnect.framework

The following header files are included in the AppConnect.framework:

TABLE 2. HEADER FILES IN APPCONNECT.FRAMEWORK (IN ALPHABETICAL ORDER)

Header file	Description and related topics
ACCompatibility.h	Header file for compatibility of AppConnect constants with Swift. **Related topics** Using the AppConnect framework in a Swift app
ACDerivedCredential.h	Described in Derived credential header files.
ACDerivedCredentialService.h	Described in Derived credential header files.
ACError.h	Defines the error domain and error codes used by the SDK's

TABLE 2. HEADER FILES IN APPCONNECT.FRAMEWORK (IN ALPHABETICAL ORDER) (CONT.)

Header file	Description and related topics
	POSIX-style APIs, and Objective-C secure file subclasses and categories. **Related topics** Secure file I/O API details
ACFileHandle.h	Defines a NSFileHandle subclass for secure file I/O. **Related topics** Secure file I/O API details
ACFileHandle.h+ACSharedSecureFile.h	Defines a category for sharing secure files with another AppConnect app. **Related topics** Secure file I/O API details
ACSecureFile.h	• Defines the POSIX-style secure file I/O APIs. • Defines `ACSecureFileLastError()` for getting more detailed error information about the POSIX-style secure file I/O APIs. **Related topics** Secure file I/O API details
ACSensitiveData.h	Defines the classes for using heightened security for sensitive data such as encryption keys. **Related topics** Securing sensitive data such as encryption keys
ACTypes.h	Defines AppConnect typedef enumerations used in AppConnectInterface.h.
ACUnwrappedFile.h	Defines the class for a host app to use to unwrap a secure file shared by an extension. **Related topics** Sharing secure files from an extension
ACWrappedAppKey.h	Defines the class for a provider app to use to create an encryption key for encrypting shared files in its extension.

TABLE 2. HEADER FILES IN APPCONNECT.FRAMEWORK (IN ALPHABETICAL ORDER) (CONT.)

Header file	Description and related topics
	Related topics Sharing secure files from an extension
ACWrappedFileReadHandle.h	Defines the class for a host app to use to get the file handle of an extension's shared, wrapped file. **Related topics** Sharing secure files from an extension
AppConnect.h	Umbrella header file for the AppConnect.framework, importing all the header files in the framework.
AppConnect+Networking.h	Defines the following APIs: • APIs for receiving upload progress for HTTP/S requests that use the AppTunnel feature. • APIs for AppTunnel diagnostics **Related topics** • Getting upload status for tunneled HTTP/S requests • AppTunnel diagnostic API details
AppConnectInterface.h	• Defines the AppConnect interface that your app uses to get configuration and security-related information from the AppConnect library. • Defines the AppConnectDelegate protocol that you implement to receive notifications from the AppConnect library of changes to this information. **Related topics** • The AppConnect interface • AppConnect-related notifications
AppConnectUIApplication.h	Defines the UIApplication subclass that the AppConnect library uses. An app imports this header file only if it uses a subclass of UIApplication. **Related topics** • Use AppConnect's UIApplication subclass • AppConnectUIApplication class

TABLE 2. HEADER FILES IN APPCONNECT.FRAMEWORK (IN ALPHABETICAL ORDER) (CONT.)

Header file	Description and related topics
NSArray+ACSecureFile.h	Defines NSArray category interfaces for secure file I/O. **Related topics** Secure file I/O API details
NSData+ACSecureFile.h	Defines NSData category interfaces for secure file I/O. **Related topics** Secure file I/O API details
NSData+ACSharedSecureFile.h	Defines NSData category interfaces for secure file I/O when sharing data among AppConnect apps. **Related topics** Secure file I/O API details
NSDictionary+ACSecureFile.h	Defines NSDictionary category interfaces for secure file I/O. **Related topics** Secure file I/O API details
NSFileManager+ACSecureFile.h	Defines NSFileManager category interfaces for secure file I/O. **Related topics** Secure file I/O API details
NSKeyedArchiver+ACSecureFile.h	Defines NSKeyedArchiver category interfaces for secure file I/O. **Related topics** Secure file I/O API details

TABLE 2. HEADER FILES IN APPCONNECT.FRAMEWORK (IN ALPHABETICAL ORDER) (CONT.)

Header file	Description and related topics
NSKeyedUnarchiver+ACSecureFile.h	Defines NSKeyedUnarchiver category interfaces for secure file operations. **Related topics** Secure file I/O API details
NSMutableArray+ACSecureFile.h	Defines NSMutableArray category interfaces for secure file I/O. **Related topics** Secure file I/O API details
NSMutableDictionary+ACSecureFile.h	Defines NSMutableDictionary category interfaces for secure file I/O. **Related topics** Secure file I/O API details

Header files in AppConnectExtension.framework

The following header files are included in the AppConnectExtension.framework:

TABLE 3. HEADER FILES IN APPCONNECTEXTENSION.FRAMEWORK (IN ALPHABETICAL ORDER)

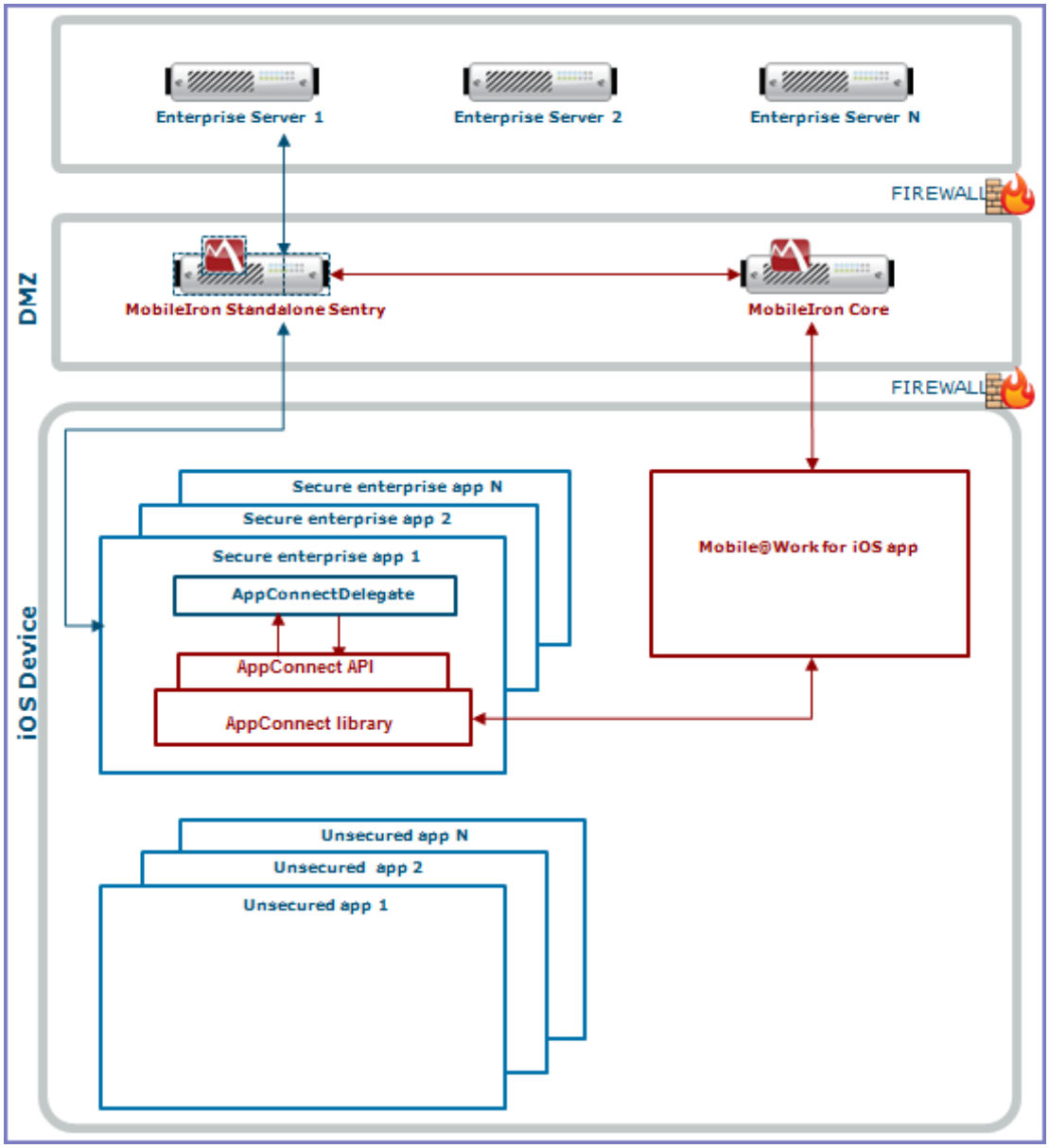
Header file	Description and related topics
ACWrappedFile.h	Defines the ACWrappedFile class used by extensions to share secure files. **Related topics** Coding the extension to share files with the host app
AppConnectExtension.h	Umbrella header file for the AppConnectExtension.framework, importing all the header files in the framework. **Related topics** Coding the extension to share files with the host app
AppConnectExtensionInterface.h	Defines AppConnectExtensionInterface class and AppConnectExtensionInterfaceProtocol. **Related topics** Coding the extension to share files with the host app

AppConnect for iOS architecture

Your app, using the AppConnect library, interacts with the MobileIron client app. The MobileIron client app is either Mobile@Work for iOS, MobileIron Go for iOS, or MobileIron AppStation for iOS. Mobile@Work interacts with Core and MobileIron Go interacts with MobileIron Cloud. AppStation is used in certain use cases instead of MobileIron Go to interact with MobileIron Cloud when a MobileIron Cloud tenant is set up for Mobile Apps Management (MAM) but not Mobile Device Management (MDM). The AppConnect library also interacts with Standalone Sentry for AppTunnel support.

The following diagram illustrates the interactions between an AppConnect app, the AppConnect library, the MobileIron server, the MobileIron client, and the Standalone Sentry. The diagram uses MobileIron Core for the server and Mobile@Work for the client.

FIGURE 1. APPCONNECT APP INTERACTION



Note The Following:

- Each secure enterprise app communicates with an AppConnect singleton object, which contains the AppConnect library.
- The AppConnect library communicates with the MobileIron client app.
- The app uses the AppConnect API to get management and security-related information, such as whether the server administrator has authorized the app to run on the device.
- Each secure enterprise app creates an object that implements the AppConnectDelegate protocol. This object receives notifications from the AppConnect library. These notifications tell the app about changes to management and security-related information.

- The MobileIron client app communicates with the MobileIron server to get management and security-related information.
  The MobileIron server provides security and management for an enterprise's devices, and for the apps and data on those devices. An administrator configures the security and management features using a web portal.
- The AppConnect object interacts with a Standalone Sentry if it is tunneling network connections to an enterprise server behind the firewall.

The MobileIron client app and AppConnect apps

The MobileIron client app supports AppConnect apps, including the following tasks:

- It communicates with the MobileIron server to get management and security-related information and passes the information to the AppConnect apps.
  The MobileIron client app periodically does an app checkin with the MobileIron server to get this information. The administrator configures the app checkin interval on the MobileIron server. It is the maximum time between app checkins while an AppConnect app is running.
- It enforces the AppConnect passcode or Touch ID/Face ID.
  The MobileIron client app prompts the device user to create an AppConnect passcode or Touch ID/Face ID when first launching any AppConnect app. The administrator configures an auto-lock timeout on the MobileIron server. After this period of inactivity, the MobileIron client app prompts the device user to reenter his AppConnect passcode or Touch ID/Face ID.

When you run your AppConnect app, the MobileIron client app sometimes automatically launches to support app checkin and the AppConnect passcode or Touch ID/Face ID. Understanding the MobileIron client app expected behavior can help you when you test your AppConnect app.

App checkin and the MobileIron client app

On each app checkin, the MobileIron client app gets AppConnect policy updates for all the AppConnect apps that have already run on the device. These updates include changes to data loss prevention policies, password settings, app configurations, and AppTunnel settings.

For example, for Mobile@Work, these updates are due to changes on MobileIron Core to:

- the AppConnect global policy for the device.
- AppConnect container policies for each of the AppConnect apps that have run on the device.
- AppConnect app configurations for each of the AppConnect apps that have run on the device.
- the current authorization status for each of the AppConnect apps that have run on the device.

The MobileIron client app does an app checkin in the following situations:

- The device user launches an AppConnect app for the first time.
  In this situation, the MobileIron client app finds out about the app for the first time, and adds it to the set of AppConnect apps for which it gets updates.
- The app checkin interval expires while an AppConnect app is running.
- The app checkin interval expired while no AppConnect apps were running and then the device user launches an AppConnect app.

In each of these situations, the MobileIron client app launches, and the device user sees the MobileIron client app momentarily. Once the MobileIron client app has completed the app checkin, the device user automatically returns to the AppConnect app.

The auto-lock timeout and the MobileIron client app

The MobileIron client app launches to prompt the device user for the AppConnect passcode or Touch ID/Face ID in the following situations:

- The auto-lock (inactivity) timeout expires while the device is running an AppConnect app and the AppConnect passcode, or Touch ID/Face ID, is the login mechanism.

If the device user is *interacting with* the app, the auto-lock timeout does not expire. This case occurs only when the device user has not touched the device for the duration of the timeout interval.

- The device user used the MobileIron client app to log out of AppConnect apps, and then launches an AppConnect app.
- The server administrator has changed the complexity rules of the AppConnect passcode, and an app checkin occurs.

In each of these situations, the MobileIron client app launches, and presents the device user with a screen for entering his AppConnect passcode or Touch ID/Face ID. After the device user enters the passcode or Touch ID/Face ID, the device user automatically returns to the AppConnect app.

Product versions required

To develop and deploy an app that uses AppConnect for iOS, you need certain products. MobileIron supports a set of product versions, and another set of product versions are compatible with apps built with this version of the AppConnect for iOS SDK.

- **Supported product versions:** The functionality of the product and version with currently supported releases was systematically tested as part of the current release and, therefore, will be supported.
- **Compatible product versions:** The functionality of the product and version with currently supported releases has not been systematically tested as part of the current release, and therefore not supported. Based on previous testing (if applicable), the product and version is expected to function with currently supported releases.

The following table summarizes supported and compatible product versions. This information is current at the time of this release. For MobileIron product versions released after this release, see that product version's release notes for the most current support and compatibility information.

TABLE 4. SUPPORTED AND COMPATIBLE PRODUCT VERSIONS FOR APPCONNECT SDK APPS

Product	Supported versions	Compatible versions
iOS	11.0 - 13.5.1	9.0 and lower are not supported
Xcode (for building apps that use the AppConnect for iOS SDK)	11	11 See also, APG-1154 in AppConnect 4.5.0 for iOS SDK revision history.
MobileIron Core and Connected Cloud	10.5.0.0, 10.6.0.0 , 10.7.0.0	10.3.0.0 - 10.4.0.0
Standalone Sentry	9.7.3, 9.8.1	9.5.0-9.6.0
Mobile@Work for iOS	12.2.2, 12.3.0	12.0.0 -12.1.0
MobileIron Cloud	70	Not applicable
MobileIron Go	5.4.0	4.0.0 - 5.3.0
MobileIron AppStation	1.3.0	Not applicable

IMPORTANT: Some AppConnect features depend on the version of MobileIron Core, MobileIron Cloud, Standalone Sentry, and the MobileIron client app.

Securing and managing the app using the AppConnect library

A MobileIron server administrator configures how mobile device users can use secure enterprise applications. The administrator sets the following app-related settings that impact your app's behavior:

- Authorization
- AppConnect passcode and Touch ID/Face ID policy
- Configuration specific to the app
- AppTunnel
- Certificate authentication to enterprise services
- Data loss prevention policies
- Custom keyboard control
- The log level and whether to log to files, described in Log messages

Additionally, the AppConnect library provides the following capabilities for your app:

- Data protection

  The AppConnect library uses encryption to protect AppConnect-related data. You can choose to protect your app's sensitive data using AppConnect for iOS APIs.

- iOS active state change notifications due to AppConnect control switches
- APIs for diagnosing AppTunnel configuration and usage

  See AppTunnel diagnostic API details.

The following steps show the flow of information from the MobileIron server to your app:

1. The MobileIron server administrator decides which app-related settings to apply to a device or set of devices.
2. The MobileIron server sends the information to the MobileIron client app.
3. The MobileIron client app passes the information to the AppConnect library. The MobileIron client app and the AppConnect library enforce the AppConnect passcode policy. The AppConnect library enforces tunneling.
4. Using the AppConnect for iOS API, your app can find out the current settings and receive notifications of changes.

Your app is responsible for:

- enforcing authorization
- handling the data loss prevention policies
- using the configuration specific to the app.
- protecting the app's data independent of device level encryption by using the AppConnect secure file I/O APIs
- logging messages to the console using the AppConnect logging APIs
- preserving the app's state when control switches to the MobileIron client app and then back to the app due to the AppConnect app check-in interval or auto-lock time expiring.

Authorization

Your app uses the AppConnect library to get the user's authorization status for using the app and to be notified of changes. For more information, see Authorization API details.

The MobileIron server administrator determines:

- whether or not each device user is authorized to use each secure enterprise app.

  If the user is not authorized, the app should not allow the user to access any secure data or functionality. If the app handles only secure data and functionality, then the app does nothing more than display a message that the user is not authorized to use the app.

- the situations that cause an authorized device user to become unauthorized.

These situations include, for example, when the device OS is compromised. the MobileIron client app reports device information to the MobileIron server. The server then determines whether to change the user to unauthorized based on security policies on Core.

When a user becomes unauthorized, the app should stop allowing the user access to any secure data or functionality.

- the situations that retire the app.

  Retiring an app means that the user is not authorized to use it, and the app removes all secure data associated with the app.

When an app is retired, you remove all its secure data. When a user is unauthorized but the app is not retired, you do not allow the user to access the data, but you do not have to remove it. The reason is that an unauthorized user can become authorized again, and therefore the secure data should become available again.

AppConnect passcode and Touch ID/Face ID policy

The AppConnect library and the MobileIron client app enforce the passcode or Touch ID/Face ID, and the auto-lock timeout. The only task for your app is to include **Privacy - Face ID Usage Description** in your app's info.plist. Beyond that plist addition, *your app does not handle the AppConnect passcode or Touch ID/Face ID at all.*

The MobileIron server administrator determines:

- whether the AppConnect passcode or Touch ID/Face ID is required, which requires the device user to enter a passcode or Touch ID/Face ID to access any secure enterprise apps.
- the complexity of the AppConnect passcode.
- the auto-lock (inactivity) timeout for the AppConnect passcode or Touch ID/Face ID.

The AppConnect library and the MobileIron client app enforce an AppConnect passcode or Touch ID/Face ID as follows:

- The MobileIron server notifies the MobileIron client app when the server administrator has enabled an AppConnect passcode or Touch ID/Face ID. The MobileIron client app prompts the user to set the passcode, if required, the next time that the device user launches or switches to a secure enterprise app.
- The user is prompted to enter the passcode or Touch ID/Face ID when the user subsequently launches or switches to a secure enterprise app but the auto-lock timeout has expired.
- The user is prompted to enter the passcode or Touch ID/Face ID when the auto-lock timeout expires *while* the user is running a secure enterprise app.
- The MobileIron client app prompts the user to set the passcode, if required, the next time the device user launches or switches to a secure enterprise app after the MobileIron server has notified the MobileIron client app that the passcode's complexity rules have changed.

Configuration specific to the app

Sometimes an app requires app-specific configuration. Some examples are:

- the address of a server that the app interacts with
- whether particular features of the app are enabled for the user
- user-related information from LDAP, such as the user's ID and password
- certificates for authenticating the user to the server that the app interacts with

You determine the app-specific configuration that your app requires. Each configurable item is a key-value pair. Each key and value is a string. A MobileIron server administrator specifies the key-value pairs for each app on the server. The administrator applies the appropriate set of key-value pairs to a set of devices. Sometimes more than one set of key-value pairs exists on the server for an app if different users require different configurations. For example, the administrator can assign a different server address to users in Europe than to users in the United States.

NOTE: When the value is a certificate, the value contains the base64-encoded contents of the certificate, which is a SCEP or PKCS-12 certificate. If the certificate is password encoded, the server automatically sends another key-value pair. The key's name is the string *<name of key for certificate>*_MI_CERT_PW. The value is the certificate's password.

Your app uses the AppConnect library to get the configuration and to be notified of changes. Then your app applies the configuration according to its requirements.

For more information, see App-specific configuration API details .

AppTunnel

Using MobileIron's AppTunnel feature, a secure enterprise app can securely tunnel HTTP and HTTPS network connections from the app to servers behind an organization's firewall. A Standalone Sentry is necessary to support AppTunnel with HTTP/S tunneling. The MobileIron server administrator handles all HTTP/S tunneling configuration on the server. Once the administrator has configured tunneling for the app on the server, the AppConnect library, the MobileIron client app, and a Standalone Sentry handle tunneling for the app.

Your app accesses its enterprise servers as it normally would using NSURLConnection or NSURLSession. Your app typically does not take any special actions related to tunneling. Although your app uses a server address that results in tunneling, your app does not know that tunneling is occurring. Typically, the MobileIron server administrator uses AppConnect's app-specific configuration to specify the enterprise server URL that the app uses. See Configuration specific to the app.

NOTE: Initialize the AppConnect library before registering any NSURLProtocol subclasses that your app uses. See Initialize the AppConnect library on page 52.

Consider the following information to ensure that your app can successfully tunnel network connections:

- AppTunnel supports only NSURLConnection and NSURLSession
- Accessing sockets directly does not use AppTunnel
- App's responsibilities when using AppTunnel

- AppTunnel supports redirects and authentication requests on HTTP/S upload
- AppTunnel with TCP tunneling

AppTunnel supports only NSURLConnection and NSURLSession

Always access servers using NSURLConnection or NSURLSession.

Note The Following:

- AppTunnel with HTTP/S tunneling does not support using NSURLSession in a background session. The traffic does not reach its destination
- You can also use networking libraries that use NSURLConnection or NSURLSession. For example, apps can use AFNetworking 3.0 because it uses NSURLSession.
- AppTunnel with HTTP/S tunneling does not support WKWebView objects.

Accessing sockets directly does not use AppTunnel

AppTunnel with HTTP/S tunneling is not supported if the app:

- accesses sockets directly.
- uses APIs that access sockets directly.

In these cases, the app cannot access a host behind the enterprise's firewall using AppTunnel with HTTP/S tunneling.

For example, AppTunnel with HTTP/S tunneling is not supported with the following APIs:

- Apple's reachability APIs that detect network and host connectivity.
- CFNetwork APIs
- ASIHTTPRequest

NOTE: Network connections using sockets for TCP connections can tunnel data by using AppTunnel with TCP tunneling. See AppTunnel with TCP tunneling.

App's responsibilities when using AppTunnel

For many apps, the app takes no special actions to use AppTunnel with HTTP/S tunneling. However, special actions are required if your app requires:

- Cached responses for URL requests that use AppTunnel with HTTP/S tunneling.
  To allow cached responses, see Caching tunneled URL responses.
- Upload status for HTTP/S requests that use AppTunnel with HTTP/S tunneling.
  Use the APIs described in Getting upload status for tunneled HTTP/S requests.

AppTunnel supports redirects and authentication requests on HTTP/S upload

When an app uses AppTunnel with HTTP/S tunneling, AppTunnel handles the following HTTP/S upload scenarios without any special actions by the app:

- HTTP/S redirect responses from the server (HTTP/S 3XX status code).
  If a server redirects an HTTP/S upload request (tunneled or not) to another URL that the MobileIron server administrator has configured for tunneling, the request is tunneled.
- Authentication required response from the server (HTTP/S 401 status code).
  The AppTunnel feature handles sending a second HTTP/S request with authentication credentials.

AppTunnel with TCP tunneling

AppTunnel can tunnel TCP traffic between an app and a server behind the company's firewall. AppTunnel with TCP tunneling does not require an app to be an AppConnect app; both AppConnect apps and standard apps can use AppTunnel with TCP tunneling.The MobileIron server administrator configures AppTunnel with TCP tunneling, including installing MobileIron Tunnel (an iOS app) on the device. *Your app takes no actions related to using AppTunnel with TCP tunneling.*

Certificate authentication to enterprise services

Without any development, an AppConnect app can send a certificate to identify and authenticate the app user to an enterprise service when the app uses an HTTPS connection. The MobileIron server administrator configures on the server which certificate for the app to use, and which connections use it. The AppConnect library, which is part of every AppConnect app, makes sure the connection uses the certificate. *Your app takes no action at all.*

Supported networking methods

Certificate authentication to enterprise services is supported only if your app uses one of the following to access the enterprise service:

- NSURLConnection
- NSURLSession

Certificate authentication to enterprise services does not support using NSURLSession in a background session.

- Networking libraries that use NSURLConnection or NSURLSession.
- UIWebView

Unsupported networking methods

Certificate authentication to enterprise services using other networking methods is not supported. For example, the following are not supported:

- accessing sockets directly
- WKWebView and other APIs that access sockets directly
  For example, these APIs are not supported: CFNetwork, ASIHTTPRequest, and Apple's reachability APIs that detect network and host connectivity.

Data loss prevention policies

An app can leak data if it uses iOS features such as copying to the iOS pasteboard, document interaction (Open In and Open From), and print capabilities. A MobileIron server administrator specifies on the server whether each app is allowed to use each of these features.

Specifically:

- The print policy indicates whether the app is allowed to use: AirPrint, any future iOS printing feature, any current or future third-party libraries or apps that provide printing capabilities.
  Your app enforces the print policy by enabling or disabling printing capabilities based on the print policy.
- The pasteboard policy specifies whether your app is allowed to copy content *to* the iOS pasteboard. If copying content is allowed, the policy specifies whether all apps, or only AppConnect apps, can paste the copied content *from* the pasteboard.
  The AppConnect library enforces the pasteboard policy. Your app disables or enables any special user interfaces that allow copying.
- The drag and drop policy specifies whether AppConnect apps can drag content to all other apps, to only other AppConnect apps, or not at all.
  The AppConnect library enforces this policy. When the policy allows dragging content to only other AppConnect apps, the AppConnect library notifies your app when the device user attempts to drag content to a non-AppConnect app. Your app can then notify the device user of the situation.
- The Open In policy specifies the apps, including the extensions that apps provide, with which your app can share documents. The policy specifies no apps, all apps, all AppConnect apps, or a set of apps. A set of apps is called the whitelist. Whether your app can share documents with the native iOS mail app is also controlled by the Open In policy.
  The AppConnect library enforces the Open In policy. Your app disables or enables any special user interfaces that give the user the option to use Open In.
- The Open From policy specifies the apps, including the extensions that apps provide, from which your app can receive documents when the other app uses the Open In iOS feature. The policy specifies no apps, all apps, all AppConnect apps, or a set of apps. A set of apps is called the whitelist.
  The AppConnect library enforces the Open From policy. Your app informs the user about the Open From policy if you want to.

The administrator applies the appropriate policies to a set of devices. Sometimes more than one set of policies exists on the MobileIron server for an app if different users require different policies.

Your app uses the AppConnect library to get the data loss prevention policies and to be notified of changes. Then your app handles the policies according to its requirements.

For more information, see:

- Pasteboard policy API details
- Drag and drop policy API details
- Open In policy API details
- Open From policy API details
- Print policy API details

Custom keyboard control

Custom keyboard extensions sometimes send data to servers when a device user enters data into an app. They send this data for assistance with word-prediction, for example. To stop this potentially harmful data loss, the MobileIron server administrator configures whether custom keyboards are allowed for an app by setting a key-value pair in the app's configuration. The key is called MI_AC_IOS_ALLOW_CUSTOM_KEYBOARDS. The key-value pair is consumed by the AppConnect library; your app does not receive it.

When the key is present, the AppConnect library controls custom keyboard use according to the key's value. If the value is true, the AppConnect library allows the AppConnect app to use custom keyboards. If the value is false, the AppConnect library does not allow custom keyboard use.

If the server administrator does not include the key-value pair for your app, the AppConnect library does not allow the app to use custom keyboards.

Related topics

Reject custom keyboard control

Data protection

AppConnect-related data

The MobileIron client app and the AppConnect library work together to use encryption to protect AppConnect-related data, such as configurations and certificates, on the device.

The encryption key is not stored on the device. It is either:

- Derived from the device user's AppConnect passcode.
- Protected by the device passcode if the administrator does not require an AppConnect passcode.
- Protected by the device passcode if the device user uses Touch ID/Face ID to access AppConnect apps.

If no AppConnect passcode or device passcode exists, the data is encrypted, but the encryption key is not protected by either passcode.

Your app does not handle data protection for AppConnect-related data. the MobileIron client app and the AppConnect library provide this data protection.

App data files

You can protect the contents of your app's data files using secure file I/O APIs provided by the AppConnect for iOS SDK. This secure file I/O capability means that data encryption for your app's data, like the AppConnect-related data, is not dependent on the device having a device passcode.

Like the AppConnect-related data, the encryption key is not stored on the device. It is either:

- Derived from the device user's AppConnect passcode.
- Protected by the device passcode if the administrator does not require an AppConnect passcode.
- Protected by the device passcode if the device user uses Touch ID/Face ID to access AppConnect apps.

The administrator can require an AppConnect passcode, a device passcode, both passcodes, or neither. The administrator can also allow a device user to use Touch ID/Face ID to access AppConnect apps, which requires a device passcode to work. By using the secure file I/O APIs, you know that your app's data is protected to the extent that the administrator requires. If your app instead relies on iOS data protection to protect your app's data, data is not protected on devices that have no device passcode. Devices having no device passcode are not uncommon when employees use their own devices at work.

NOTE: When using secure file I/O APIs, the existence of a file, its file name, its path, its approximate size, its creation date, and its last modification date are not encrypted. Any of these items possibly reveal sensitive information.

The following table summarizes the protection of the data that AppConnect apps save on the device. Note that if a device user uses Touch ID/Face ID to access AppConnect apps, a device passcode is available.

TABLE 5. DATA ENCRYPTION OF APP DATA

	Device passcode but no AppConnect passcode	AppConnect passcode but no device passcode	Device passcode and AppConnect passcode	Neither a device passcode or AppConnect passcode
SDK apps that enable iOS data protection (typical behavior)	App data encrypted	iOS encrypts the data, but the encryption key is not protected.	App data encrypted	iOS encrypts the data, but the encryption key is not protected.
SDK apps that use SDK-provided secure file I/O	App data encrypted	App data encrypted	App data encrypted	iOS encrypts the data, but the encryption key is not protected.

Some of the secure file I/O APIs also support sharing the encrypted files with other AppConnect apps. These APIs rely on an additional encryption group ID to create the encryption key. Only apps which use the same encryption group ID can read the data. If you use these APIs to share encrypted files with other apps, you provide an app-specific configuration key name to the MobileIron server administrator in your app documentation. The MobileIron server administrator then provides your app and the other apps the same encryption group ID through app-specific configuration for each app.

The SDK provides the following types of secure file I/O APIs:

- POSIX-style APIs
- Objective-C subclasses
- Objective-C class categories

Before your app uses these APIs, use the AppConnect library to get the status of:

- secure services
  Currently, the only secure service is secure file I/O.
- secure file I/O

The AppConnect library notifies the app about changes in both statuses.

For more information, see Secure services API details.

NOTE: If your app uses custom cryptography, you can get encryption keys from the AppConnect library. For more information, see Encryption keys for custom cryptography.

Log messages

The AppConnect for iOS SDK provides APIs for your app to use to log messages to the Apple System Log facility, also known as the device's console. The MobileIron server administrator can specify that the messages are also

logged to files on the device.

You specify the log level of each message. The log levels are, from least verbose to most verbose:

- Error
- Warning
- Status
- Info
- Verbose
- Debug

Note The Following:

- Error, warning and status messages are always logged to the console.
- Info messages are logged to the console only if that level is specified by the MobileIron server administrator in the app-specific configuration.
- Verbose and debug messages are logged to the console only if both of the following are true:
  - The server administrator specified the level and a debug code in the app-specific configuration.
  - The device user enabled the level in the MobileIron client app using the debug code specified by the server administrator. Note that the verbose or debug levels are disabled automatically after 24 hours. The device user can manually disable them sooner.

  Because the verbose and debug levels require a debug code, you can include sensitive data in messages logged at those levels.

For details, including examples of the kinds of messages to log at each level, see Log messages API details.

Getting Started with the AppConnect for iOS SDK

Getting started tasks

Objective-C apps: Follow the instructions in Before you begin adding the AppConnect SDK to your app. Then follow the instructions in either First-time use of SDK in your app or Task lists for upgrading the SDK in your app.

Swift apps: Follow the instructions in Before you begin adding the AppConnect SDK to your app. Then follow the instructions in Using the AppConnect framework in a Swift app.

Xamarin apps: Follow the instructions in Before you begin adding the AppConnect SDK to your app. Then follow the instructions in Developing AppConnect Apps with Xamarin .

Once you have completed these tasks, your app is ready to use the AppConnect for iOS API to, for example, enforce MobileIron server settings and apply app-specific configurations from the MobileIron server.

NOTE: If your app is a Cordova app, use the AppConnect for iOS Cordova Plugin, described in the *MobileIron AppConnect for iOS Cordova Plugin Developers Guide*.

Before you begin adding the AppConnect SDK to your app

- **Download the AppConnect for iOS SDK**.
  Download the latest version of the AppConnect for iOS SDK ZIP file to your app's Xcode project folder or other convenient location. The ZIP file is available at help.mobileiron.com in the **Software** tab.
  The ZIP file is named AppConnectiOSSDK_V<*version*>_<*build*>.zip where:
  - <*version*> is the version number of the SDK
  - <*build*> is the build number of the SDK.
- **Verify required product versions.**
  Be sure you have the required product versions for working with apps built with the AppConnect for iOS SDK.
  See Product versions required .
- **Support fast app switching.**
  Make sure your app supports fast app switching. Fast app switching means that the app can go into the background for a short time without iOS terminating it. The AppConnect for iOS SDK requires that apps support this feature. Most apps support fast app switching.
  To ensure that your app supports fast app switching, in your app's Info.plist, remove the `UIApplicationExitsOnSuspend` key if it is present.
  NOTE: Your app does not need to support any of the `UIBackgroundModes`, such as `audio` or `voip`

First-time use of SDK in your app

If you are adding the AppConnect for iOS SDK to your app for the first time, do the tasks in Getting started task list.

Task lists for upgrading the SDK in your app

To upgrade an app that uses a prior version of the AppConnect for iOS SDK to use SDK version 4.3.0, use the appropriate task list in the following table.

TABLE 6. UPGRADE TASK LIST

SDK version from which you are upgrading	Upgrade task list
SDK 4.2.1	• Replace the AppConnect.framework bundle in the project folder. • If you are using the AppConnectExtension.framework, replace the AppConnectExtension.framework bundle in the project folder. • Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme. See Declare the AppConnect URL schemes as allowed.
SDK 4.2.0	• Replace the AppConnect.framework bundle in the project folder. • If you are using the AppConnectExtension.framework, replace the AppConnectExtension.framework bundle in the project folder. • Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme. See Declare the AppConnect URL schemes as allowed.
SDK 4.1.1	• Replace the AppConnect.framework bundle in the project folder. • If you are using the AppConnectExtension.framework, replace the AppConnectExtension.framework bundle in the project folder. • Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme. See Declare the AppConnect URL schemes as allowed.
SDK 4.1	• Replace the AppConnect.framework bundle in the project folder. • If you are using the AppConnectExtension.framework, replace the AppConnectExtension.framework bundle in the project folder. • Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme. See Declare the AppConnect URL schemes as allowed.
SDK 4.0	• Replace the AppConnect.framework bundle in the project folder. • If you are using the AppConnectExtension.framework, replace the AppConnectExtension.framework bundle in the project folder. • Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme. See Declare the AppConnect URL schemes as allowed.
SDK 3.1 through 3.5	1. Remove the existing AppConnect.framework and AppConnectResources.bundle from your Xcode project.

TABLE 6. UPGRADE TASK LIST (CONT.)

SDK version from which you are upgrading	Upgrade task list
	2. Remove the libProtocolsBuffer.a and libCrypto.a libraries from your Xcode project if you added them only for making your app an AppConnect app. However, if you use specific versions of these libraries for other reasons, or indirectly link to versions of these libraries, keep them in your project and make sure they are linked **before** the AppConnect.framework. The libCrypto.a that is part of the AppConnect.framework is FIPS compliant. Therefore, if your only reason for linking in your own libCrypto.a is to be FIPS compliant, you can remove it. 3. Remove the following command from your Xcode project's **Other Linker Flags** (in **Linking** under **Build Setting**: `-force_load $(SRCROOT)/AppConnect.framework/AppConnect` 4. Make sure **Other Linker Flags** include `-ObjC` because the AppConnect.framework is an Objective-C framework. 5. Navigate to AppConnect.framework in the top-level of the extracted AppConnect SDK directory. 6. Drag and drop AppConnect.framework to **Embedded Binaries** in the **General** settings of your Xcode project's target. When Xcode prompts you to choose options for adding the file, select **Create groups**. 7. Navigate to AppConnectResources.bundle in the AppConnect.framework directory of the extracted AppConnect SDK directory. 8. Drag and drop AppConnectResources.bundle to **Copy Bundle Resources** in the **Build Phases** settings of your Xcode project's target. When Xcode prompts you to choose options for adding the file, select **Create groups**. 9. Add a **Run Script** section to the **Build Phases** settings of your Xcode project's target. 10. Add post_embed_actions.sh, located in the top-level of the extracted AppConnect SDK directory, to the scripts to run. This script removes extra architectures from the AppConnect app's binary. Removing desktop architectures is required before submitting your app to the Apple App Store. 11. Make sure **Enable Bitcode** is set to **No** in **Build Options** in the **Build Settings** of your Xcode project's target. 12. Include the boolean key MI_AC_PROVIDE_SCREEN_BLUR set to YES in your app's info.plist. For details, see Add AppConnect-related entries to your Info.plist.

TABLE 6. UPGRADE TASK LIST (CONT.)

SDK version from which you are upgrading	Upgrade task list
	13. Include **Privacy - Face ID Usage Description** to your app's info.plist, with a string value indicating the purpose of Face ID use. For example, add the value **AppConnect**. If you manually add this key, its name is NSFaceIDUsageDescription. 14. Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme. See Declare the AppConnect URL schemes as allowed.
SDK 2.4 through 3.0	1. Do the upgrade steps for SDK 3.1 through 3.5. 2. Recommended: If your app is a dual-mode app, modify the dual mode behavior to include `+shouldStartAppConnect:`. For details, see Developing Third-party Dual-mode Apps. 3. Recommended: If your app enforced the Open In data loss prevention policy, note that the AppConnect library now enforces the policy. Make appropriate code modifications. For details, see Open In policy API details .
SDK 1.9.1 through 2.3.1	1. Do the steps for SDK 2.4 through 3.0. 2. Declare the AppConnect URL schemes as allowed
SDK 1.9.0 or earlier	Refer to the 3.5 version of this guide, available at MobileIron AppConnect 3.5 for iOS SDK App Developers Guide or contact MobileIron Technical Support.

Getting started task list

NOTE: If you have an app that already uses a prior version of the AppConnect for iOS SDK and want to upgrade the app to use the current SDK version, see Task lists for upgrading the SDK in your app.

If you are adding the AppConnect for iOS SDK to your app for the first time, do the following tasks:

1. Add AppConnect files and settings to your Xcode project
2. Add your own libcrypto.a, libProtocolBuffers.a, and libssl.a libraries if needed
3. Register as a handler of the AppConnect URL scheme
4. Declare the AppConnect URL schemes as allowed
5. Add AppConnect-related entries to your Info.plist
6. Use AppConnect's UIApplication subclass
7. Initialize the AppConnect library
8. Wait for the AppConnect singleton to be ready
9. Optional: Specify app permissions and configuration in a plist file.

Add AppConnect files and settings to your Xcode project

Do the following tasks to add AppConnect files and settings to your app's Xcode project:

1. Make sure **Other Linker Flags** include `-ObjC` because the AppConnect.framework is an Objective-C framework.
2. Navigate to AppConnect.framework in the top-level of the extracted AppConnect SDK directory.
3. Drag and drop AppConnect.framework to **Embedded Binaries** in the **General** settings of your Xcode project's target.
   When Xcode prompts you to choose options for adding the file, select **Create groups**.
4. Navigate to AppConnectResources.bundle in the AppConnect.framework directory of the extracted AppConnect SDK directory.
5. Drag and drop AppConnectResources.bundle to **Copy Bundle Resources** in the **Build Phases** settings of your Xcode project's target.
   When Xcode prompts you to choose options for adding the file, select **Create groups**.
6. Add a **Run Script** section to the **Build Phases** settings of your Xcode project's target.
7. Add post_embed_actions.sh, located in the top-level of the extracted AppConnect SDK directory, to the scripts to run.
   This script removes extra architectures from the AppConnect app's binary. Removing desktop architectures is required before submitting your app to the Apple App Store.
8. Make sure **Enable Bitcode** is set to **No** in **Build Options** in the **Build Settings** of your Xcode project's target.

Add your own libcrypto.a, libProtocolBuffers.a, and libssl.a libraries if needed

The AppConnect.framework includes:

- the libcrypto.a library.
  The included libcrypto.a library is FIPS compliant.
- the libProtocolBuffers.a library.
  The included libProtocolBuffers.a library is by Booyah, Inc.
- the libssl.a library.

However, if you need specific versions of these libraries, you can add them to your Xcode project.

When you add one of these libraries to your Xcode project, make sure it is listed ***higher than*** AppConnect.framework in your Xcode project in the **General** tab under **Linked Frameworks and Libraries**.

Alternatively, you can make sure your version of one of these libraries is used by adding a `-force_load` command to the linker in Xcode. For example. in your Xcode project, in **Build Settings**, under **Linking**, in **Other Linker Flags**, add:

```
-force_load "$(PROJECT_DIR)/dependencies/openssl_fips_ios/lib/libcrypto.a"
```

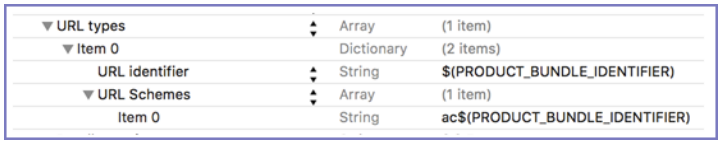
Register as a handler of the AppConnect URL scheme

Your app must handle the AppConnect URL scheme. Mobile@Work, MobileIron Go, and MobileIron AppStation use this URL scheme to communicate with your app's instance of the AppConnect library.

Register the AppConnect URL scheme by modifying the app's Info.plist. Edit the key called **URL types** so that:

- the sub-item **URL identifier** has the value of your app's bundle identifier
- the sub-item **URL Schemes**' sub-item 0 has the value **ac** concatenated with your app's bundle identifier

These key-value pairs are illustrated by the following excerpt from HelloAppConnect's HelloAppConnect-Info.plist:



NOTE: If you are editing the Info.plist file directly, it should include the following:

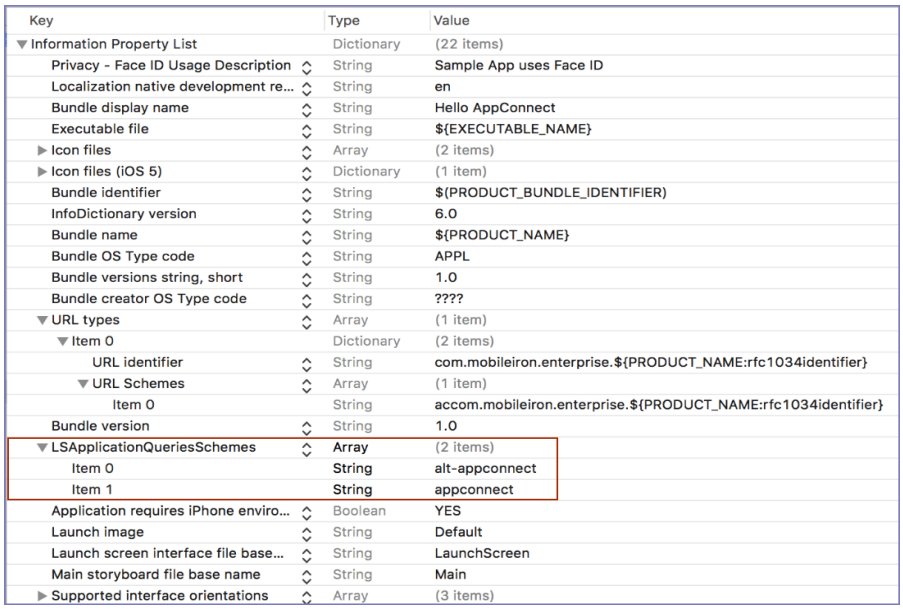
```
<key>CFBundleURLTypes</key>
<array>
   <dict>
      <key>CFBundleURLSchemes</key>
      <array>
         <string>ac$(PRODUCT_BUNDLE_IDENTIFIER)</string>
      </array>
   </dict>
</array>
```

Declare the AppConnect URL schemes as allowed

Declare the `appconnect` and the `alt-appconnect`URL schemes in your app's Info.plist as allowed URL schemes. Your app's instance of the AppConnect library:

- uses the `appconnect` URL scheme to communicate with Mobile@Work or MobileIron Go.
- uses the `alt-appconnect` URL scheme to communicate with MobileIron AppStation.

To allow the `appconnect` and `alt-appconnect` URL schemes, add a key called `LSApplicationQueriesSchemes` as shown in this example from HelloAppConnect's HelloAppConnect-Info.plist:



Key	Type	Value
Information Property List	Dictionary	(22 items)
Privacy - Face ID Usage Description	String	Sample App uses Face ID
Localization native development re...	String	en
Bundle display name	String	Hello AppConnect
Executable file	String	${EXECUTABLE_NAME}
Icon files	Array	(2 items)
Icon files (iOS 5)	Dictionary	(1 item)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
URL types	Array	(1 item)
Item 0	Dictionary	(2 items)
URL identifier	String	com.mobileiron.enterprise.${PRODUCT_NAME:rfc1034identifier}
URL Schemes	Array	(1 item)
Item 0	String	accom.mobileiron.enterprise.${PRODUCT_NAME:rfc1034identifier}
Bundle version	String	1.0
LSApplicationQueriesSchemes	Array	(2 items)
Item 0	String	alt-appconnect
Item 1	String	appconnect
Application requires iPhone enviro...	Boolean	YES
Launch image	String	Default
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Supported interface orientations	Array	(3 items)

Add AppConnect-related entries to your Info.plist

- Enable screen blurring
- Allow Face ID

Enable screen blurring

The AppConnect library can automatically blur your app's screen whenever it is not active. This security measure protects the app's data from being captured in screenshots. The AppConnect library blurs the screen when `-applicationWillResignActive:` is called and unblurs it when `-applicationDidBecomeActive:` is called.

To enable screen blurring, add the key MI_AC_PROVIDE_SCREEN_BLUR to your app's Info.plist as a Boolean. Set the value to YES.

When you set the Info.plist key MI_AC_PROVIDE_SCREEN_BLUR to YES, the MobileIron server administrators can disable screen blurring by setting a key-value pair on the server for your app's configuration. The server key is MI_AC_ENABLE_SCREEN_BLURRING with the value false.

NOTE: If you already implemented screen blurring in your app, remove that code and use the MI_AC_PROVIDE_SCREEN_BLUR plist key. Using the plist key ensures that all AppConnect apps behave consistently.

Allow Face ID

Include **Privacy - Face ID Usage Description** to your app's info.plist, with a string value indicating the purpose of Face ID use. For example, add the value **AppConnect**. If you manually add this key, its name is NSFaceIDUsageDescription.

Server administrators can allow the use of Touch ID or Face ID instead of an AppConnect passcode. Therefore, this Info.plist entry is required on iOS 11 through the most recently released version as supported by MobileIron.

Use AppConnect's UIApplication subclass

To use AppConnect's UIApplication subclass:

1. Open main.m for editing.
2. Add the following line to your import statements:
   `#import "AppConnect/AppConnect.h"`
3. Change the third argument of the call to `UIApplicationMain()` to `kACUIApplicationClassName`.
   The third argument, the `principalClassName` argument, is the UIApplication class or subclass for the app. For example, in the HelloAppConnect app provided with the AppConnect for iOS SDK, the statement that calls `UIApplicationMain` is:

```
return UIApplicationMain(argc, argv, kACUIApplicationClassName,

                         NSStringFromClass([AppDelegate class]));
```

NOTE: If you use a subclass of UIApplication for your app, see Using your own UIApplication subclass.

Initialize the AppConnect library

To initialize the AppConnect library for your app to use:

1. Open your AppDelegate source file and header file for editing.
2. Add the following line to your import statements in your AppDelegate header file:
   `#import "AppConnect/AppConnect.h"`
3. Create a class that implements the AppConnectDelegate protocol.
   Usually this class is also the AppDelegate for your app. For example, in AppDelegate.h in HelloAppConnect, the AppDelegate class implements the AppConnectDelegate protocol.
   `@interface AppDelegate : UIResponder <UIApplicationDelegate, AppConnectDelegate>`
   Some of the methods of the AppConnectDelegate protocol are optional. Implement only the optional methods that relate to your app's functionality.
4. Call the static method `+initWithDelegate:` of the AppConnect class.The method takes as a parameter an object of the class that implements the AppConnectDelegate protocol.
   For example, in HelloAppConnect, in the AppDelegate class implementation, the method `-application:didFinishLaunchingWithOptions:` calls `+initWithDelegate:` as follows:
   `[AppConnect initWithDelegate:self];`
   NOTE: If the class that implements the AppConnectDelegate protocol is not your AppDelegate, pass an instance of that class instead of `self`.
5. Save the singleton instance of the AppConnect library.

For example, in HelloAppConnect, the AppDelegate object saves the singleton instance in the appConnect property:

```
[self setAppConnect:[AppConnect sharedInstance]];
```

6. Call the AppConnect singleton's method `-startWithLaunchOptions:`.
   The app must call this method from its AppDelegate's method `-application:didFinishLaunchingWithOptions:`, and must pass along its `launchOptions` parameter value.
   For example, in HelloAppConnect:
   ```
   [self.appConnect startWithLaunchOptions:launchOptions];
   ```
   After this step, the AppConnect singleton is initializing. However, the app cannot yet use the singleton's instance properties. The app can:
   - use the AppConnect class properties.
   - use the methods of the AppConnect singleton object.
   - register any NSURLProtocol subclasses that the app uses.

   If your app uses AppTunnel with HTTP/S tunneling, be sure this NSURLProtocol registration occurs after initializing the AppConnect library.
7. If your application supports UIScene, call the method `-sceneWillConnectToSessionWithOptions`.
   The app must call the method from its UISceneDelegate's method `-scene:willConnectToSession:options:`, and must pass along the UIScene connection options as input parameter to the AppConnect instance method `-sceneWillConnectToSessionWithOptions:`.
   Example
   ```
   @implementation MySceneDelegate
   - (void)scene:(UIScene *)scene willConnectToSession:(UISceneSession *)session
   options:(UISceneConnectionOptions *)connectionOptions {
   [self.appConnect sceneWillConnectToSessionWithOptions:connectionOptions];
   }
   @end
   ```
8. Indicate in the user interface that the app is initializing if the app requires the AppConnect singleton's instance properties to determine what to do. For example, use an activity indicator (spinner). Remove the indication after the app is notified that the AppConnect singleton is ready.
   One reason this indication is important involves when to display sensitive data. Do not show any sensitive data until the AppConnect singleton is ready, because until that time, the app cannot determine whether it is authorized. Only an authorized app should show sensitive data.

Wait for the AppConnect singleton to be ready

The app cannot use the AppConnect singleton's instance properties until the `ready` property on the AppConnect singleton is set to `YES`. It is set to `YES` when the callback method `-appConnectIsReady:` in your AppConnectDelegate protocol implementation is called. The app can now access the instance properties, such as `authState` and `pasteboardPolicy`, on the AppConnect singleton.

Before accessing any instance properties, use the `isReady` getter to make sure the properties are accessible.

For example, in HelloAppConnect, the `-appConnectIsReady:` callback method calls `-updateLabels:`. The `-updateLabels:` method calls various methods that access the instance properties on the AppConnect singleton. Because other methods also call `-updateLabels:`, the method first checks the `isReady` property:

```
if ([self.appConnect isReady]) {

    // Call methods that access instance properties.
}
else {
        authInfoText = @"Ready: NO (AppConnect is not ready yet)";
        policyInfoText = @"AppConnect is not ready yet";
        configInfoText = @"AppConnect is not ready yet";
}
```

For details about the `-appConnectIsReady:` callback method and the `ready` property, see AppConnect ready API details .

Optional: Specify app permissions and configuration in a plist file

If your app is an in-house app, you can specify default values for:

- the data loss prevention policies, such as the Open In policy
- the key-value pairs for your app-specific configuration

Specifically, you can provide a special plist file called AppConnect.plist as part of your in-house app that:

- specifies whether your app should be allowed by default to copy to the iOS pasteboard, use document interaction (Open In and Open From), and print.
- specifies app-specific configuration keys and default values.

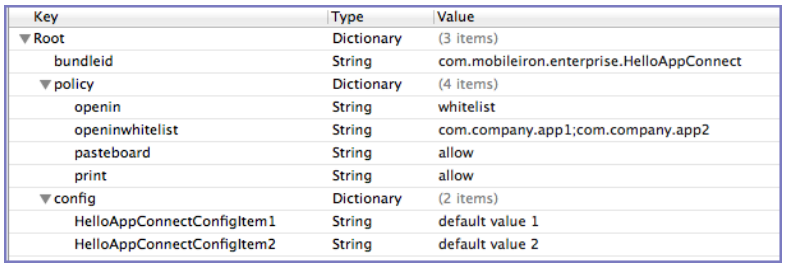
These default values are used by the MobileIron server to make it easier for the server administrator to set up your app with the correct data loss prevention policies and app-specific configurations. *Your app never reads the AppConnect.plist*.

When you include the AppConnect.plist in your app:

1. When an administrator uploads your in-house app to the MobileIron server, the server uses this plist file to automatically create server policies that contain your specified data loss prevention policies and app-specific configuration.
2. The administrator can then edit these policies.
   For example:
   - If one of your app-specific configuration keys requires a URL of an enterprise server, the administrator provides that value.
   - If the administrator requires stricter data loss prevention policies than your app's default values, the administrator changes the values.
3. The administrator then applies these policies to the appropriate set of devices.

4. When your app runs, it receives the data loss prevention policies and app-specific configuration by using the AppConnect for iOS APIs, described in AppConnect for iOS API.
   For example, to handle app-specific configurations, you use the `config` property (an `NSDictionary` object) and the callback method `-appConnect:configChangedTo:`
5. If the administrator later changes the data loss prevention policies or app-specific configuration, your app receives the updates by using the AppConnect for iOS APIs.

An example of an AppConnect.plist file as viewed in Xcode looks like the following:



Key	Type	Value
▼ Root	Dictionary	(3 items)
bundleid	String	com.mobileiron.enterprise.HelloAppConnect
▼ policy	Dictionary	(4 items)
openin	String	whitelist
openinwhitelist	String	com.company.app1;com.company.app2
pasteboard	String	allow
print	String	allow
▼ config	Dictionary	(2 items)
HelloAppConnectConfigItem1	String	default value 1
HelloAppConnectConfigItem2	String	default value 2

To set up an AppConnect.plist file:

1. Create a plist file called AppConnect.plist.
2. Place it in the root directory of your app.
3. In the Root key of AppConnect.plist, place a key called `bundleid` with the type String, and set the value to the bundle ID of your app.
4. In the Root key of AppConnect.plist, create two keys called `policy` and `config`, each with the type Dictionary.
5. In the `policy` dictionary, create keys called `openin`, `openinwhitelist`, `openfrom`, `openfromwhitelist`,`pasteboard`, and `print`, each with the type String.
6. Set these keys' values as given in the following table:

TABLE 7. APPCONNECT.PLIST KEYS AND VALUES

Key	Possible values and meanings
`openin`	• `allow` Document interaction is allowed with all other apps. • `disable` Document interaction is not allowed. • `whitelist` Only documents in the `openinwhitelist` list can open documents from your app. • `appconnect` Document interaction is allowed with all other AppConnect apps. This value results in the app receiving a whitelist in the Open In policy API. The whitelist contains the list of all currently authorized AppConnect apps. You do not enter an `openinwhitelist` key in the plist. See The openInPolicy and openInWhitelist properties on page 92.
`openinwhitelist`	Semi-colon separated list of the bundle IDs of the apps with which document interaction is allowed. This key is necessary when the `openin` key has the value `whitelist`.
`pasteboard`	• `allow` Pasteboard interaction is allowed with all other apps. That is, this option allows the device user to be able to copy content from your app to the iOS pasteboard. Then, any app can copy from the content from the pasteboard. • `disable` Pasteboard interaction is not allowed. • `appconnect` Pasteboard interaction is allowed only with other AppConnect apps. That is, this option allows the device user to be able to copy content from your app to the iOS pasteboard. Then, only other AppConnect apps can copy the content from the pasteboard.
`print`	• `allow` Printing is allowed. • `disable` Printing is not allowed.

7. In the `config` dictionary, create keys as required for your app.
8. Optionally, add values for the keys. The values must be String types.

NOTE: The value `$USERID$` in the example tells Core to substitute the device user's user ID for the value. Other possible variables are `$EMAIL$` and `$PASSWORD$`. Depending on the Core configuration, custom variables called `$USER_CUSTOM1$` through `$USER_CUSTOM4$` are sometimes available.

Using your own UIApplication subclass

If your app uses its own subclass of UIApplication, do the following:

1. Derive your subclass from AppConnectUIApplication instead of UIApplication.
   You will need the following import statement:
   `#import "AppConnect/AppConnectUIApplication.h"`
2. Change the third argument of the call to `UIApplicationMain()` to the name of your subclass of AppConnectUIApplication.
   The third argument, the `principalClassName` argument, is the UIApplication subclass for the app.
3. When you override an UIApplication method in your subclass, always invoke the method implementation of the superclass AppConnectUIApplication at the end of your method.
   For example:
   `[super sendEvent:event]`
   If you do not invoke the superclass implementation, AppConnect features will not work in your app.

Using the AppConnect framework in a Swift app

- First time use of SDK in your Swift app
- Tasks for upgrading the SDK in your Swift app

First time use of SDK in your Swift app

The following procedure describes what to do to add the AppConnect framework to a Swift app.

NOTE: When you add the AppConnect framework into your Xcode project, the Swift interfaces corresponding to all the Objective-C APIs are automatically generated by Xcode.

Before you begin

Do the tasks in Before you begin adding the AppConnect SDK to your app.

Procedure

1. Do the following steps from the Getting started task list:
   a. Add AppConnect files and settings to your Xcode project.
   b. Add your own libcrypto.a, libProtocolBuffers.a, and libssl.a libraries if needed.
   c. Register as a handler of the AppConnect URL scheme.
   d. Declare the AppConnect URL schemes as allowed.
   e. Add AppConnect-related entries to your Info.plist.
   f. Optional: Specify app permissions and configuration in a plist file

2. Add a file named main.swift to your Xcode project, if you don't already have one.
3. In main.swift, add the following code:

```
import Foundation
import AppConnect

UIApplicationMain(
    CommandLine.argc,
    UnsafeMutableRawPointer(CommandLine.unsafeArgv)
        .bindMemory(
            to: UnsafeMutablePointer<Int8>.self,
            capacity: Int(CommandLine.argc)),
    ACUIApplicationClassName,
    NSStringFromClass(YourAppDelegate.self)
)
```

4. Add a bridging header file, if you don't already have one, to your Xcode project. Name the file:
   <app name>-Bridging-Header.h
   Example:
   HelloSwiftAppConnect-Bridging-Header.h
5. In the bridging header file, import AppConnect.h:
   `#import <AppConnect/AppConnect.h>`
6. Go to your Xcode project's **Build Settings** for the Swift app target. Under **Swift Compiler - General**, set **Objective-C Bridging Header** to the bridging header file, including the path.
7. Create a class that implements the AppConnectDelegate protocol. Usually this class is also the AppDelegate for your app.
8. Initialize the AppConnect class with your AppConnectDelegate, save the singleton instance of the AppConnect library, and initialize the AppConnect library. Then wait for the initialization to complete. The following code is an excerpt from HelloSwiftAppConnect in the file HSAppDelegate.swift:

```
import UIKit
import AppConnect

class HSAppDelegate: UIResponder, UIApplicationDelegate, AppConnectHandler {

       var appConnect: AppConnect?

       func application(_ application: UIApplication,
               didFinishLaunchingWithOptions launchOptions:
               [UIApplicationLaunchOptionsKey : Any]? = nil) -> Bool {

               AppConnect.log(at: .status, message: "HelloAppConnect started")
               self.startAppConnect(launchOptions: launchOptions)
                       return true
       }

       func startAppConnect(launchOptions: [AnyHashable : Any]? = [:]) {
```

```
            AppConnect.initWith(self)
            self.appConnect = AppConnect.sharedInstance()
            self.appConnect!.start(launchOptions: launchOptions)

            // Wait for appConnectIsReady() before using any of the AppConnect
            // singleton's instance properties. The app can use AppConnect class properties
            // and methods of the AppConnect singleton object.
            // If your app uses AppTunnel with HTTP/S tunneling, be sure to register any
            // NSURLProtocol subclasses AFTER initializing the AppConnect library.

            // Indicate in the user interface that the app is initializing if the app requires
            // the AppConnect singleton's instance properties to determine what to do. For example,
            // use an activity indicator (spinner). Remove the indication after the app is notified
            // that the AppConnect singleton is ready.
            // One reason this indication is important involves when to display sensitive data. Do
            // not show any sensitive data until the AppConnect singleton is ready, because until
            // that time, the app cannot determine whether it is authorized. Only an authorized app
            //should show sensitive data.
        }

        func appConnectIsReady(_ appConnect: AppConnect) {
            // The app can now use the AppConnect singleton's instance properties.

            self.updateLabels()
        }

}
```

If your application supports UIScene, call the method `sceneWillConnectToSession(with:)`.

The app must call the method from its UISceneDelegate's method `scene(_:willConnectTo:options:)`, and must pass along the UIScene connection options as input parameter to the AppConnect instance method `sceneWillConnectToSession(with:)`.

Example:

```
class MySceneDelegate: UIResponder, UIWindowSceneDelegate {
       func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options con-
nectionOptions: UIScene.ConnectionOptions){
            AppConnect.sharedInstance()?.sceneWillConnectToSession(with: connectionOptions)
       }
}
```

Tasks for upgrading the SDK in your Swift app

If you are upgrading your Swift app from a previous version of the AppConnect for iOS SDK:

- Replace the AppConnect.framework bundle in the project folder.
- If you are using the AppConnectExtension.framework, replace the AppConnectExtension.framework bundle in the project folder.

Troubleshooting

AppConnect(ACURLSessionDataDelegateProxy.o)' does not contain bitcode.

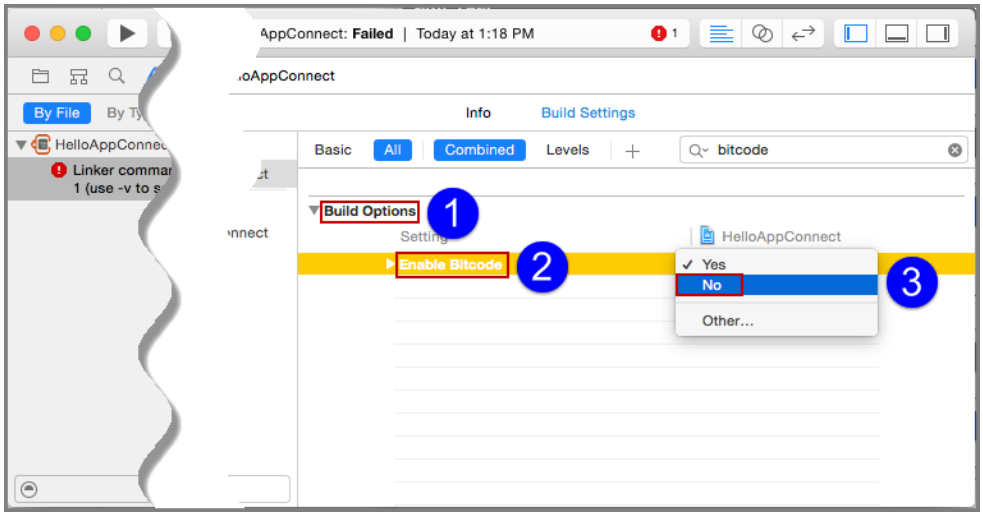
Problem: Bitcode is enabled in build options, but should be disabled.

When you build your project, the following error occurs:

```
AppConnect(ACURLSessionDataDelegateProxy.o)' does not contain bitcode. You must rebuild it
with bitcode enabled (Xcode setting ENABLE_BITCODE), obtain an updated library from the
vendor, or disable bitcode for this target. for architecture arm64
```

Solution:

Disable Bitcode in the project's Build Options, for example:



Lexical or preprocessor issue when building your app

Problem: path missing in #import statement

When you build your project, the following compiler error occurs:

```
Lexical or Preprocessor Issue:
'AppConnect.h' file not found
```

Solution

Be sure your #import statements include the path to AppConnect.h and other header files included in AppConnect.framework. For example:

```
#import "AppConnect/AppConnect.h"
```

App cannot start because AppConnectResources.bundle not found

Problem

Your app crashes immediately on launch with the following error message:

```
[AppConnect:Error] AppConnect is unable to start because AppConnectResources.bundle was not
found in the app.
*** Terminating app due to uncaught exception 'AppConnect unable to start', reason:
'AppConnectResources.bundle was not found in the app
```

Solution

Make sure you have added AppConnectResources.bundle from AppConnect.framework to your app. See Add AppConnect files and settings to your Xcode project.

App crashes in call to -startWithLaunchOptions:

Problem

Your app crashes immediately on launch, in the call to the AppConnect singleton's method `-startWithLaunchOptions:`.

When this error occurs, the AppConnect library:

- logs an error.
  `@"AppConnect error: AppConnect is unable to start because [UIApplication sharedApplication] is not an instance AppConnectUIApplication."`
- throws an NSException object. The object's `name` method returns the string `"AppConnect unable to start"`. The object's `reason` method returns the string `"[UIApplication sharedApplication] is not an instance of AppConnectUIApplication."`

Solution

The call in main.m to the function `UIApplicationMain` is incorrect. Follow the instructions in Use AppConnect's UIApplication subclass.

Application error: Unable to communicate with the application

Problem

The MobileIron client app displays this error message:

Application error: Unable to communicate with the application. Please contact the application developer. The application's bundle ID is <*your application's bundle ID*>.

Solution

This error occurs when the AppConnect library tries to contact the MobileIron client app, but you did not register the app as a handler for the AppConnect URL scheme.

See Register as a handler of the AppConnect URL scheme on page 50.

App crashes due to uncaught ACPropertyAccessException

Problem

Your app crashes due to the following uncaught exception:

```
<Error>: *** Terminating app due to uncaught exception 'ACPropertyAccessException', reason:
'Method -[AppConnect_impl <method name>] called before recovering the first unlock key'
```

The AppConnect library throws this exception if the app accesses the instance properties on the AppConnect singleton before the AppConnect singleton is ready.

Solution

Refactor your code to make sure you check the AppConnect singleton getter `isReady` before accessing any instance properties. If `isReady` is `YES`, you can access the instance properties. If `isReady` is `NO`, wait for the AppConnect library to call the callback method `-appConnectIsReady:` before accessing the properties.

See AppConnect ready API details on page 80.

Developing Third-party Dual-mode Apps

What is a dual-mode app?

If your AppConnect app is distributed from the Apple App Store, due to Apple App Store requirements, your app is required to work as either:

- an AppConnect app for enterprise users
- a regular app for general consumers

Such an app is called a *dual-mode* app. Using one code base and APIs in the AppConnect for iOS SDK, the app automatically decides when it launches which mode to behave in:

- As an AppConnect app
  The app supports the AppConnect features, such as authorization, data loss prevention, and secure file I/O. MobileIron, through the MobileIron server, the MobileIron client app, and the AppConnect library, provides AppConnect management.
- As a regular app
  The app supports none of the AppConnect features. Furthermore, depending on the app, the functionality available as a regular app can differ significantly from the functionality available as an AppConnect app. For example, as a regular app, the app does not allow the user to access any sensitive enterprise data.
  A typical reason that an app runs as a regular app is that AppConnect is not configured for the device on the MobileIron server. An app also runs as a regular app if the MobileIron client app has not yet been installed on the device.

AppConnect apps distributed from the Apple App Store must be dual-mode apps. If you are a third-party app developer, you typically build apps for Apple App Store distribution. If you are an in-house app developer, your apps are typically distributed from the MobileIron server.

IMPORTANT: If your app is not distributed from the Apple App Store and works only as an AppConnect app, ignore the dual-mode capability and associated APIs.

Dual-mode sample app

A sample app that shows proper dual-mode app behavior is included with the AppConnect for iOS SDK zip file. The app illustrates how and when to use the AppConnect for iOS APIs related to dual-mode behavior. It also illustrates using AppConnect capabilities and secure data only when behaving as an AppConnect app.

The app is a simple note-taking app that allows the user to create a set of notes. The app uses the model-view-controller design pattern. The model classes are Notes and Policies. The view controllers are NotesViewController, NoteDetailViewController, SettingsViewController and AuthMessageViewController. The class DualModeAppDelegate is the main app controller.

The following table summarizes the files:

TABLE 8. DUAL-MODE SAMPLE APP FILES

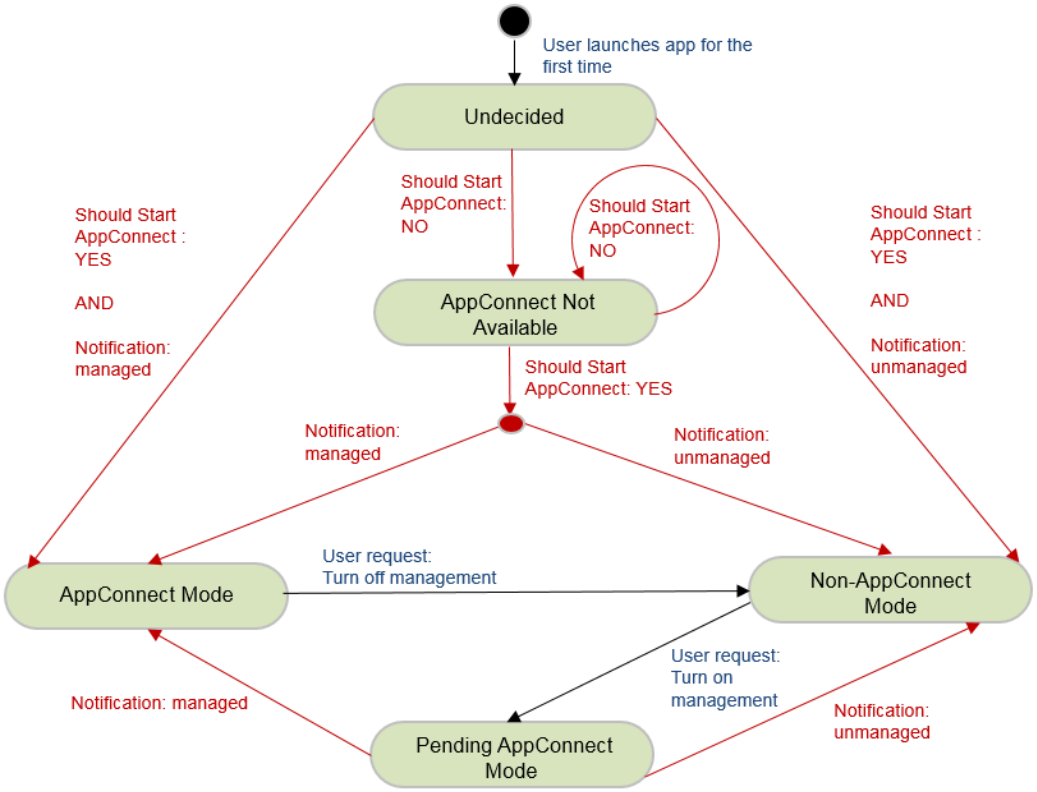
File	Description
DualModeAppDelegate.h/m	UIApplicationDelegate for the app
Notes.h/m	• Handles application logic for adding and retrieving notes. • Uses secure file I/O when required.
Policies.h/m	• Implements AppConnectDelegate • **Handles the dual-mode state transitions.** • Keeps track of whether to use secure file I/O and handle DLP policies.
NotesViewController.h/m	Provides user interface for showing the list of notes.
NotesDetailViewController.h/m	Provides user interface for showing the contents of an individual note.
SettingsViewController.h/m	Provides user interface for changing between AppConnect app behavior and regular app behavior.
AuthMessageViewController.h/m	Provides user interface for displaying authorization status messages. NOTE: Displaying the authorization status is only applicable when behaving as an AppConnect app.

Dual-mode app states

An app must maintain a dual-mode state that indicates whether it is behaving as an AppConnect app or a regular app. It stores this state persistently, so that when it next launches, it knows how to behave. The possible states are:

- **Undecided**
  The app has initialized for the first time and has not yet decided whether to run in **AppConnect Mode** or **Non-AppConnect Mode**.
- **AppConnect Mode**
  The app is running as an AppConnect app. It supports the AppConnect features, such as authorization, data loss prevention, and secure file I/O.
  In **AppConnect Mode**, the app can change state only to **Non-AppConnect Mode**, and only if the app user requests the change using a user interface provided by the app.
- **Non-AppConnect Mode**
  The app is running as a regular app.
  In **Non-AppConnect Mode**, the app can change state only to **AppConnect Mode**, and only if the app user requests the change using a user interface provided by the app.
- **Pending AppConnect Mode**
  The app changes to this state if the device user explicitly requests a change from **Non-AppConnect Mode** to **AppConnect Mode** using the app's user interface. For example, device users in an enterprise sometimes have installed and used an app before the enterprise requires it as an AppConnect app. In this state, the app is waiting for a notification from the AppConnect library to find out whether MobileIron AppConnect components are managing the app.
- **AppConnect Not Available**
  AppConnect is not yet available on the device because the MobileIron client app is not yet installed on the device. The app runs as a regular app. If the MobileIron client app is later installed, on subsequent launches the app will decide whether to run in **AppConnect Mode** or **Non-AppConnect Mode**.
  It is important for an app to delay its decision to run in **AppConnect Mode** or **Non-AppConnect Mode** until after the MobileIron client app is installed. The reason is that users often launch the app before installing the MobileIron client app. If the app decides on **Non-AppConnect Mode**, it can not leave that state without user actions, such as using an app-provided user interface, or re-installing the app. The **AppConnect Not Available** state allows apps to automatically change to **AppConnect Mode** when re-launched after the MobileIron client app is installed.

The following diagram summarizes the state transitions that a dual-mode app implements. See High-level dual-mode app behavior for more information about these state transitions.



Data encryption states

A dual-mode app encrypts its data only if all of the following are true:

- The app is in **AppConnect Mode**.
- Secure services are available.

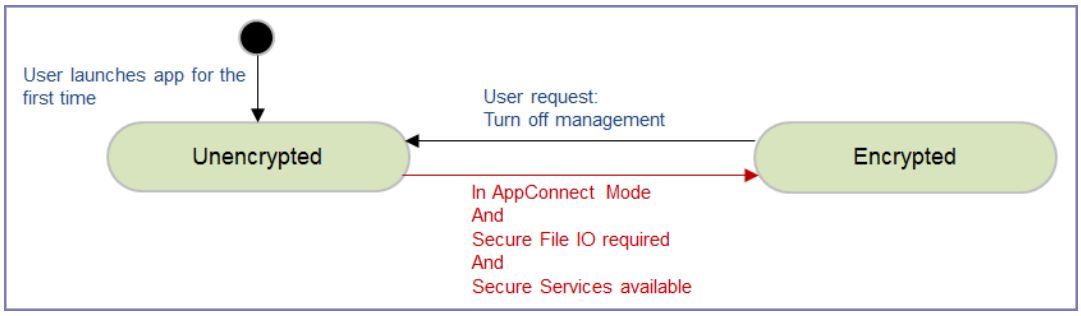
Secure services are available only when the app's authorization status is authorized.

- The secure file I/O policy requires secure file I/O.

Therefore, the app maintains a data encryption state. It stores this state persistently, so that when it next launches, it knows how to behave. The possible states are:

- **Unencrypted**
  The app does not encrypt its data.
- **Encrypted**
  The app encrypts its data.

The following diagram summarizes the state transitions that a dual-mode app implements. See High-level dual-mode app behavior for information about these state transitions.



The state change to **Encrypted** state depends on three conditions: that the app is managed, that secure services are available, and that the secure file I/O policy requires secure file I/O. Because the order of these notifications can vary, upon receiving any of the notifications, the app checks if all three conditions are true yet.

For an example of checking whether to change to the **Encrypted** state, see the DualMode sample app's method `-checkEncryptionState:` in Policies.m.

Actions when changing to the Encrypted state

When changing to the **Encrypted** state, the app starts using secure file I/O APIs for new sensitive data. Also, the app determines what to do with existing unencrypted data.

Consider these options for existing data:

- Secure existing sensitive data.
  Your app can use the secure file I/O APIs to encrypt existing sensitive data. The dual-mode sample app provides an example of this behavior.
  MobileIron recommends this option for sensitive data. If device users upgrade from a previous version of your app to a new dual-mode version, this option ensures that they do not lose data.
  However, some data can remain unsecured. For example, user display preferences are typically not sensitive information.
- Remove existing data.
  Your app can remove existing data if doing so does not cause disruption to the app users.

Actions when changing to the Unencrypted state

When changing to the **Unencrypted** state, the app removes all sensitive data.

High-level dual-mode app behavior

When the app launches for the first time

When a dual-mode app launches for the first time, it does not know whether it is managed by MobileIron. It does the following high-level steps:

1. Sets its initial dual-mode state to **Undecided**.
2. Sets its initial encryption state to **Unencrypted**.

3. Checks whether AppConnect is available.
   - If AppConnect is not available, the app changes its dual-mode state to **AppConnect Not Available**, and continues as a regular app.
   - If AppConnect is available, the app starts the AppConnect library.
4. Waits for a notification from the AppConnect library indicating whether MobileIron is managing the app.
5. Changes its dual-mode state to **AppConnect Mode** or **Non-AppConnect Mode** according to the notification.
   - When changing to **Non-AppConnect Mode**, the app notifies the AppConnect library that it is retiring. Normally, the MobileIron server decides when to retire an app. In this case, the app is retiring itself. Then the app stops the AppConnect library. It behaves as a regular app.
   - When changing to **AppConnect Mode**, the app behaves as an AppConnect app. However, the app changes its data encryption state to **Encrypted** only if secure apps are available and the secure file I/O policy requires secure file I/O. The app uses the data encryption state to determine whether it can use secure file I/O APIs.
6. Stores both the dual-mode state and data encryption state persistently for the next time it launches.

NOTE: For more details, including specific API calls for these steps, see API call sequence when the app launches.

When an app subsequently launches

On subsequent launches, the app does the following high-level steps:

1. Gets the dual-mode state and data encryption state that it stored.
2. Checks the dual-mode state, and takes the following actions depending on the state.
   a. **AppConnect Mode**: Starts the AppConnect library.
   The app continues as an AppConnect app. It uses the data encryption state to determine whether it can use secure file I/O APIs.
   b. **Non-AppConnect Mode**: Continues as a regular app.
   The app does not start the AppConnect library.
   c. **AppConnect Not Available**: Checks whether AppConnect is available.
      - If AppConnect is not available, the app stays in **AppConnect Not Available**, and continues as a regular app.
      - If AppConnect is available, the app starts the AppConnect library, and waits for a notification indicating whether MobileIron is managing the app.
3. After receiving the notification, changes its dual-mode state to **AppConnect Mode** or **Non-AppConnect Mode** according to the notification.
   - When changing to **Non-AppConnect Mode**, the app notifies the AppConnect library that it is retiring. Normally, the MobileIron server decides when to retire an app. In this case, the app is retiring itself. Then the app stops the AppConnect library. It behaves as a regular app.
   - When changing to **AppConnect Mode**, the app behaves as an AppConnect app. However, the app changes its data encryption state to **Encrypted** only if secure apps are available and the secure file I/O policy requires secure file I/O. The app uses the data encryption state to determine whether it can use secure file I/O APIs.
4. Stores both the dual-mode state and data encryption state persistently for the next time it launches.

NOTE: For more details, including specific API calls for these steps, see API call sequence when the app launches.

User requests to switch to Non-AppConnect Mode

A dual-mode app can provide a user interface that allows the device user to explicitly request that MobileIron no longer manage the app. That is, the user requests a change to **Non-AppConnect Mode**. This user interface can be useful if a device user leaves an enterprise, but still wants to use the app as a regular app.

Users are typically not aware of the term “AppConnect”. Therefore, the user interface should use other terminology. The dual-mode sample app uses “Managed by MobileIron” in its user interface. Another possibility is “Secure enterprise mode”.

When switching from **AppConnect Mode** to **Non-AppConnect Mode**, the app does the following high-level steps:

1. Removes all its secure data, since regular apps do not have secure data.
2. Sets the data encryption state to **Unencrypted**, and stores it persistently for the next time it launches.
3. Notifies the AppConnect library that it is retiring.
   Normally, the MobileIron server decides when to retire an app. In this case, the app is retiring itself.
4. Stops the AppConnect library.
5. Stores its dual-mode state, **Non-AppConnect Mode**, persistently for the next time it launches.
6. Continues running as a regular app.
   For example, the app no longer enforces AppConnect policies or uses AppConnect features such as secure file I/O.

NOTE: For more details, including specific API calls for these steps, see API call sequence when user requests Non-AppConnect Mode.

User requests to switch to AppConnect Mode

A dual-mode app can provide a user interface that allows the device user to explicitly request that MobileIron manage the app. That is, the user requests a change to **AppConnect Mode**. For example, device users in an enterprise sometimes have installed and used an app before the enterprise requires it as an AppConnect app.

Users are typically not aware of the term “AppConnect”. Therefore, the user interface should use other terminology. The dual-mode sample app uses “Managed by MobileIron” in its user interface. Another possibility is “Secure enterprise mode”.

When switching from **Non-AppConnect Mode** to **AppConnect Mode**, the app does the following high-level steps:

1. Starts the AppConnect library.
2. Changes to the **Pending AppConnect Mode** state.
3. Waits for a notification from the AppConnect library indicating that MobileIron is managing the app.
4. If the app receives the notification that MobileIron is managing the app, the app changes state to **AppConnect Mode**, and persistently stores the new state. It begins behaving as an AppConnect app. For example, it enforces DLP policies.
   If secure services are available and the secure file I/O policy requires secure file I/O, the app changes the encryption state to **Encrypted**. The app decides what to do with existing data as described in Actions when changing to the Encrypted state.

NOTE: For more details, including specific API calls for these steps, see API call sequence when user requests AppConnect Mode.

Data loss prevention policy handling

When a dual-mode app changes from **Non-AppConnect Mode** to **AppConnect Mode**, it handles the AppConnect data loss prevention policies that it supports. For example, if the app supports the Open In policy, based on the policy it receives from the AppConnect library, it enables or disables any Open In user interfaces. When changing to **Non-AppConnect Mode**, the app stops handling the AppConnect DLP policies.

Dual-mode API details

The AppConnect for iOS API provides properties and methods that allow an app to behave as a dual-mode app.

The ACManagedPolicy enumeration

The `ACManagedPolicy` enumeration provides the possible managed policy values for the app

```
typedef enum {
    ACMANAGEDPOLICY_UNKNOWN   =  0, // The AppConnect library has not yet determined
                                    // whether the app is managed by MobileIron.
    ACMANAGEDPOLICY_UNMANAGED =  1, // The application is not currently managed by
                                    // MobileIron.
    ACMANAGEDPOLICY_MANAGED   =  2, // The application is currently managed by
                                    // MobileIron.
 } ACManagedPolicy;
```

The managedPolicy property

The read-only `managedPolicy` property on the AppConnect singleton contains an `ACManagedPolicy` value. The value reflects the current status of the managed policy for the app. The managed policy indicates whether MobileIron is managing the app.

The app can access the `managedPolicy` property only after:

- It has called the `-startWithLaunchOptions:` method on the AppConnect singleton.
- It has received the `-appConnectIsReady:` callback, that sets the `ready` property on the AppConnect singleton to `YES`.

NOTE: Currently, apps have no need to use the `managedPolicy` property. Dual-mode apps depend on notifications to instigate changes to the app's dual-mode state.

After your app starts the AppConnect library, the AppConnect library determines the managed policy value, and then:

1. updates the `managedPolicy` property.
2. calls the `-appConnect:managedPolicyChangedTo:` method to provide your app the current managed policy value.

Dual mode methods

A dual-mode app uses the following methods:

- The +shouldStartAppConnect: class method
- The -appConnect:managedPolicyChangedTo: callback method
- The stop method
- The retire method

The +shouldStartAppConnect: class method

```
+(BOOL)shouldStartAppConnect;
```

Dual mode apps call this method to determine whether to start the AppConnect library. The method returns `YES` if:

- the MobileIron client app is installed or
- the MobileIron client app had been installed but is now deleted, and the app had previously run in AppConnect Mode

This method is necessary because users often launch an app before the MobileIron client app is installed. When an app launches the first time, the app determines whether it is managed by MobileIron, and therefore determines whether to run as an AppConnect app (**AppConnect Mode**) or a regular app (**Non-AppConnect Mode).** Once an app has chosen one of these modes, it cannot change to the other mode without user actions, such as using an app-provided user interface, or re-installing the app. Therefore, an app should call `+shouldStartAppConnect:` to determine whether to delay choosing between **AppConnect Mode** and **Non-AppConnect Mode** until its next launch. If `+shouldStartAppConnect:` returns `NO`, the app delays the choice and runs as a regular app. On the app's next launch, if `+shouldStartAppConnect:` returns `YES`, it makes the choice to run as an AppConnect app without any user action.

The -appConnect:managedPolicyChangedTo: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect managedPolicyChangedTo:
                                  (ACManagedPolicy)newManagedPolicy;
```

Implement this method only if your app is a dual-mode app.

When a change occurs to the managed policy, the AppConnect library:

1. Sets the `managedPolicy` property on the AppConnect object to the new `ACManagedPolicy` value.
2. Calls the `appConnect:managedPolicyChangedTo:` method, which provides the new `ACManagedPolicy` value in its parameter.

In this method, the app changes its dual-mode state to **AppConnect Mode** or **Non-AppConnect Mode**.

The stop method

```
-(void)stop;
```

The `-stop` method on the AppConnect singleton object:

- shuts down the AppConnect library for the app.
- deallocates the AppConnect shared instance -- the `sharedInstance` static property on the AppConnect class.

The app calls the `-stop` method when it changes state to **Non-AppConnect Mode**.

If at a later time, the user requests to change to **AppConnect Mode**, the app restarts the AppConnect library.

NOTE: The app can call the `+logAtLevel:format:` and `+logAtLevel:format:args:` class methods and get the `+(ACLogLevel)logLevel` property even after calling `-stop:`. When the AppConnect library is stopped, the log level is always `ACLOGLEVEL_STATUS`.

See API call sequence when user requests AppConnect Mode for examples of:

- when to call the `-stop` method
- restarting the AppConnect library

The retire method

```
-(void)retire;
```

The `-retire` method on the AppConnect singleton object informs the AppConnect library that the app is retiring. Normally, the MobileIron server decides when to retire an app. In this case, the app is retiring itself.

Calling `-retire` causes the AppConnect library to:

- clean up information it keeps about the app, including secure data.
- set its managedPolicy status for the app to `ACMANAGEDPOLICY_UNKNOWN`.

IMPORTANT: An app must call `-retire` and then immediately call `-stop` when it is changing to **Non-AppConnect Mode**.

For an example of when to call the `-retire` method, see API call sequence when user requests AppConnect Mode.

API call sequence when the app launches

When a dual-mode app launches, it uses its dual-mode state and whether AppConnect is available to determine how to proceed. It must determine whether it is managed by MobileIron, and therefore whether to behave as an AppConnect app or a regular app.

NOTE: Dual-mode sample app code snippets illustrating this behavior are from:

- File: Policies.m
- Methods: `-initPrivateWithLaunchOptions:` and `-appConnect:managedPolicyChangedTo:`

Specifically, when launching, the app does the following:

1. Gets its persisted dual-mode state and data encryption state. On the first launch, because no persisted states exist, the app sets the dual-mode state to **Undecided** and the encryption state to **Unencrypted**.
2. Determines whether to start the AppConnect library.

```
// Do not start the AppConnect library when the dual-mode state is Non-AppConnect Mode.
// Otherwise, start it only if shouldStartAppConnect() says you should.
if (DMS_NonACMode != _state && [AppConnect shouldStartAppConnect]) {

    [AppConnect initWithDelegate:self];
    _ac = [AppConnect sharedInstance];
    [_ac startWithLaunchOptions:launchOptions];

 }
else if (DMS_Undecided == _state) {
       // Change the state to AppConnect Not Available.
       // Persistently store the state, and continue as a regular app

       [self setState:DMS_ACNotAvailable];
}
```

3. If the AppConnect library was not started**,** continue as a regular, non-AppConnect app.

In this case, if the dual-mode state had been persisted prior to this launch, it was either **Non-AppConnect Mode**, **AppConnect Not Available,** or **Pending AppConnect Mode**. If it had not been persisted, the state changed from **Undecided** to **AppConnect Not Available.**

4. If the AppConnect library was started and the persisted dual-mode state was **AppConnect Mode:**
   - Wait for the `-appConnectIsReady:` callback method before accessing any instance properties on the AppConnect singleton.
   - Continue as an AppConnect app. Regarding data encryption, if the data encryption state is **Encrypted**, it can use secure file I/O.
5. If the AppConnect library was started and the dual-mode state is **AppConnect Not Available, Undecided,** or **Pending AppConnect Mode,** wait for the AppConnect library to call the `-appConnect:managedPolicyChangedTo:` callback method.
   The state change depends on the value of the `newManagedPolicy` parameter in the callback method:
   - If the value `ACMANAGEDPOLICY_MANAGED`, the app changes to **AppConnect Mode,** and persistently stores the new dual-mode state.
     The app begins behaving as an AppConnect app. For example, it handles DLP policies, and when the data encryption state changes to **Encrypted**, it starts using secure file I/O
   - If the value `ACMANAGEDPOLICY_UNMANAGED`, the app changes to **Non-AppConnect Mode**, and persistently stores the new dual-mode state.
     It calls `-retire`, and then stops the AppConnect library:

     ```
     [_ac retire];
     [_ac stop]:
     _ac = nil;
     ```

     The app begins behaving as a regular, non-AppConnect app.

API call sequence when user requests Non-AppConnect Mode

If the device user, through the app's user interface, requests to change to **Non-AppConnect Mode**, the app makes the change.

NOTE: Dual-mode sample app code snippets illustrating this behavior are from:
- File: Policies.m
- Methods: `-switchToNonACMode:`

The app does the following:

1. Performs its usual retire actions, such as removing all its sensitive data, since regular apps do not have sensitive data.
2. Sets the data encryption state to **Unencrypted**, since regular apps do not encrypt data. It persistently saves the state.
3. Persistently saves its dual-mode state as **Non-AppConnect Mode**.
4. Calls the `-retire` method on the AppConnect singleton object, and then stops the AppConnect library.

```
[_ac retire];
[_ac stop]:
_ac = nil;
```

5. Continues as a regular, non-AppConnect app.

When the app next launches, it checks its dual-mode state. Because the state is **Non-AppConnect Mode**, the app does not start the AppConnect library.

API call sequence when user requests AppConnect Mode

If the device user, through the app's user interface, requests to change to **AppConnect Mode,** the app attempts to make the change.

NOTE: Dual-mode sample app code snippets illustrating this behavior are from:
- File: Policies.m
- Methods: `-attemptSwitchToACMode:` and `-appConnect:managedPolicyChangedTo:`

The app does the following:

1. Changes to the **Pending AppConnect Mode** state, and persistently stores the state.
2. Starts the AppConnect library, using `-startWithLaunchOptions:.`

```
[AppConnect initWithDelegate:self];
_ac = [AppConnect sharedInstance];
[_ac startWithLaunchOptions:nil];
```

NOTE: When restarting the AppConnect library, the parameter passed to `-startWithLaunchOptions:` is `nil`.

3. Waits for the AppConnect library to call `-appConnect:managedPolicyChangedTo:`.

When `-appConnect:managedPolicyChangedTo:` is called:

- If the `newManagedPolicy` parameter has the value `ACMANAGEDPOLICY_UNMANAGED:`
  The app changes its dual-mode state back to **Non-AppConnect Mode**. The app persistently stores the state.
  The app calls `-retire`, and then stops the AppConnect library:

```
[_ac retire];
[_ac stop]:
_ac = nil;
```

The app notifies the user of the failure to change to **AppConnect Mode**. It continues behaving as a regular, non-AppConnect app.

- If the `newManagedPolicy` parameter has the value `ACMANAGEDPOLICY_MANAGED:`
  The app changes its dual-mode state to **AppConnect Mode**. The app persistently stores the state.
  If secure services are available, and the secure file I/O policy is required, the app sets the data encryption state to **Encrypted**. The app decides what to do with existing data. For example, the DualMode sample app encrypts all its existing notes. As the app continues, it checks the data encryption state to determine whether to use secure file I/O APIs.
  Also, when changing to **AppConnect Mode**, the app checks if the authorization status is retired. If it is, the app performs its usual retire actions, such as removing all its sensitive data.
  Finally, the app notifies the user of the successful change to **AppConnect Mode**. It continues behaving as an AppConnect app.

AppConnect for iOS API

- The AppConnect interface
- AppConnect-related notifications
- Multithread support
- AppConnect ready API details
- Authorization API details
- App-specific configuration API details
- Pasteboard policy API details
- Drag and drop policy API details
- Open In policy API details
- Open From policy API details
- Print policy API details
- Log messages API details
- Secure services API details
- Version property
- Getting upload status for tunneled HTTP/S requests
- Caching tunneled URL responses
- AppConnectUIApplication class
- Encryption keys for custom cryptography
- Securing sensitive data such as encryption keys
- iOS active state change notifications due to AppConnect control switches
- Secure file I/O API details
- Sharing secure files from an extension
- AppTunnel diagnostic API details
- UIScene support

Related topics

- Developing Third-party Dual-mode Apps

The AppConnect interface

The AppConnect interface provides your app's primary interactions with the AppConnect library. The AppConnect interface declares static methods that your app uses to initialize its use of the AppConnect library and get the singleton instance of the AppConnect object. For details, see:

- Use AppConnect's UIApplication subclass
- Initialize the AppConnect library.

The AppConnect interface also declares the properties and methods your app uses to interact with the AppConnect library. However, the app cannot access the instance properties on the AppConnect singleton until the AppConnect singleton has completed its initialization. For details about checking when the AppConnect singleton is ready, see:

- Wait for the AppConnect singleton to be ready
- AppConnect ready API details

For details each of the AppConnect interface's properties and methods, see:

- Authorization API details
- App-specific configuration API details
- Pasteboard policy API details
- Drag and drop policy API details
- Open In policy API details
- Print policy API details
- Log messages API details
- Secure services API details
- Version property
- Caching tunneled URL responses
- Encryption keys for custom cryptography
- iOS active state change notifications due to AppConnect control switches

NOTE: The AppConnect interface also provides methods specifically for dual-mode apps. These methods are described in Developing Third-party Dual-mode Apps.

AppConnect-related notifications

Your app receives notifications about changes to:

- the ready status of the AppConnect singleton
- the user's authorization status
- app-specific configuration
- data loss prevention policies
- secure services status
- the secure file I/O policy
- the log level
- app state changes due to AppConnect events

Upon receiving a notification, your app:

1. Makes appropriate changes to its logic, display, and data.
2. In most cases, calls an API to inform the AppConnect library about its success or failure in making the changes.

Notification methods in the AppConnectDelegate protocol

Your app receives notifications by implementing the AppConnectDelegate protocol.

Your app must implement the notification callback methods for:

- handling the change to the ready status of the AppConnect singleton:

  `-appConnectIsReady:`

- handling authorization status changes:

  `-appConnect:authStateChangedTo:withMessage:`

Your app optionally implements the notification callback methods for handling app-specific configuration changes, data loss prevention policy changes, secure services changes, log level changes and changes to the app state due to AppConnect events:

```
-appConnect:configChangedTo:
-appConnect:openInPolicyChangedTo:newWhitelist:
-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:
-appConnect:openURLAttemptedWhenUnauthorizedForURL:
-appConnectAttemptedDragAndDropToNonAppConnectApp:
-appConnect:pasteboardPolicyChangedTo:
-appConnect:copyAttemptedWhenUnauthorized:
-appConnect:printPolicyChangedTo:
-appConnect:secureServicesAvailabilityChangedTo:
-appConnect:secureFileIOPolicyChangedTo:
-appConnect:logLevelChangedTo:
-applicationWillResignActiveForAppConnect:
-applicationDidBecomeActiveFromAppConnect:
```

You implement only the optional callback methods that your application needs. For example, if your application does not copy content to the iOS pasteboard, do not implement `-appConnect:pasteboardPolicyChangedTo:`.

Notification acknowledgments

Your app must inform the AppConnect library of your app's success or failure in applying changes it receives in notifications. Depending on the type of notification, your app calls one of the following methods of the AppConnect singleton object:

```
-authStateApplied:message:

-configApplied:message:

-openInPolicyApplied:message:

-pasteboardPolicyApplied:message:

-printPolicyApplied:message:

-secureFileIOPolicyApplied:message:
```

NOTE: No notification acknowledgments exist for ready status notifications, log level notifications, or notifications of app state changes due to AppConnect events.

Each method takes a parameter that is an `ACPolicyState` enumeration value:

```
typedef enum {
    ACPOLICY_UNSUPPORTED = 0, // The policy is not supported by this application
    ACPOLICY_APPLIED     = 1, // The policy was applied successfully
    ACPOLICY_ERROR       = 2, // An error occurred applying the policy
} ACPolicyState;
```

Typically, you pass either `ACPOLICY_APPLIED` or `ACPOLICY_ERROR`. If you do not implement one of the optional notification methods, the AppConnect library behaves as if your app had passed `ACPOLICY_UNSUPPORTED`.

Multithread support

Regarding multithread support:

- The AppConnect library is thread-safe. Your app can concurrently call all methods of the AppConnect singleton object from multiple threads without deadlocking, crashing, corrupting data, or providing unpredictable results. Also on concurrent calls, the methods will not block for a long time.
- Calls that the AppConnect library makes to AppConnectDelegate methods are dispatched to the delegate on the main thread.
- Each secure file API has the same multithreading capabilities, if any, that the corresponding unsecured API has. This correspondence is true for the Posix-style APIs, the methods of ACFileHandle, and the Objective-C category methods that the AppConnect SDK provides. For example, if a POSIX function locks a file, the corresponding secure function honors that file locking. Refer to the documentation of the corresponding unsecured API for specifics. In general, however, use standard practices to serialize access to a file from multiple threads.

Related topics

- The AppConnect interface
- AppConnect-related notifications
- Secure file I/O API details

AppConnect ready API details

The ready property

The AppConnect for iOS API provides a read-only property on the AppConnect singleton called `ready`:

```
@property (nonatomic, readonly, getter=isReady) BOOL ready;
```

This property and its getter method `isReady` indicate whether the AppConnect singleton is ready for the app to access the singleton's instance properties. The app can access the instance properties only if `isReady` is `YES`. If `isReady` is `NO`, an attempt to access an instance property throws an exception. The thrown NSException object has the name `"ACPropertyAccessException"`.

When the app calls the AppConnect singleton's method `-startWithLaunchOptions:`, the value of `ready` is `NO`. When the AppConnect library calls the callback method `-appConnectIsReady:`, the value changes to `YES`. The value remains `YES` for the life of the app.

Impacted instance properties

When `isReady` is `NO`, accessing the following instance properties throw an exception:

```
@property (nonatomic, readonly) ACManagedPolicy managedPolicy;
@property (nonatomic, readonly) ACAuthState authState;
@property (unsafe_unretained, nonatomic, readonly) NSString *authMessage;
@property (nonatomic, readonly) ACPasteboardPolicy pasteboardPolicy;
@property (nonatomic, readonly) ACOpenInPolicy openInPolicy;
@property (unsafe_unretained, nonatomic, readonly) NSSet *openInWhitelist;
@property (nonatomic, readonly) ACPrintPolicy printPolicy;
@property (nonatomic, readonly) ACSecureFileIOPolicy secureFileIOPolicy;
@property (unsafe_unretained, nonatomic, readonly) NSDictionary *config;
```

NOTE: You can access the instance property `secureServicesAvailability` at any time.

The -appConnectIsReady: callback method

You are required to implement this method, which is in the AppConnectDelegate protocol:

```
-(void)appConnectIsReady:(AppConnect *)appConnect;
```

The AppConnect library calls this method when the value of the `ready` property has changed. The AppConnect library calls this method one time after the app calls the AppConnect singleton's method `-startWithLaunchOptions:`. The value of `ready` is changed to `YES`, which means that the instance properties on the AppConnect singleton are initialized and ready for the app to access.

In the `-appConnectIsReady:` method:

- Access the instance properties on the AppConnect singleton.
- Update the app with the current authorization status, data loss prevention policies, secure file I/O policy, and configuration key-value pairs.
- Remove the user interface indication that informed the user that the app was initializing.

NOTE: Always update the app's policies and configuration status in the `-appConnectIsReady:` method, which the AppConnect library calls every time the app is launched. The AppConnect library calls other callback methods, such as the callback methods for authorization, data loss prevention policies, and configuration, *only if* the status has changed. Therefore, you can always expect all these callback methods on the first launch of the app. However, subsequent launches often result in the AppConnect library calling only the `-appConnectIsReady:` method.

Pseudocode for -isAppConnectReady:

The following pseudocode illustrates how to use the `isReady` getter and the `-isAppConnectReady:` callback method. In this example:

- The same class implements the UIApplicationDelegate protocol and the AppConnectDelegate protocol.
- The class has an instance property called `appConnect` for saving the AppConnect singleton.

```
- (void)applicationDidBecomeActive:(UIApplication *)application
{
        if ([self.appConnect isReady]) {
            [self updateWithAppConnectPolicies];
        }
        else {
            [self presentAppInitializingWithMessage:NSLocalizedString
                                        (@"Authorizing. Please wait...", nil)];
        }
}

-(void)appConnectIsReady:(AppConnect *)appConnect {
    [self updateWithAppConnectPolicies];
    [self dismissAppInitializing];
 }

-(void)updateAppConnectPolicies {

   // Check isReady since this method can be called from methods besides
   // -appConnectIsReady:

   if ( [appConnect isReady]) {
       // Check the app’s authorization, policies, and configuration status
       // and update the app appropriately.
   }
}
```

Authorization API details

The AppConnect for iOS API provides properties and methods that allow an app to handle the device user's authorization status for using the app. For an overview of this feature, see Authorization .

The ACAuthState enumeration

The `ACAuthState` enumeration provides the possible authorization statuses for the device user to use the application:

```
typedef enum {
    ACAUTHSTATE_UNAUTHORIZED    =  0, // The user is not authorized to access sensitive
                                      // data or views in this app.
    ACAUTHSTATE_AUTHORIZED      =  1, // This is the only state in which the user is
                                      // authorized to access sensitive data or views.
    ACAUTHSTATE_RETIRED         =  2, // The app must erase all sensitive data,
                                      // including any stored authentication
                                      // credential.
} ACAuthState;
```

The authState and authMessage properties

The following read-only properties on the AppConnect singleton relate to authorization:

TABLE 9. AUTHORIZATION PROPERTIES ON THE APPCONNECT SINGLETON

Property	Description
`authState`	An `ACAuthState` value that indicates the current authorization status of the device user for using the app.
`authMessage`	A string value that indicates the reason for the current authorization status.

When your app launches:

- Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.
- Wait for the `-appConnectIsReady:` callback method before accessing the `authState` and the `authMessage` properties.
  While waiting, indicate in the user interface that the app is initializing if the app requires the AppConnect singleton's instance properties to determine what to do. For example, use an activity indicator (spinner).
  One reason this indication is important involves when to display sensitive data. Do not show any sensitive data until the AppConnect singleton is ready, because until that time, the app cannot determine whether it is authorized. Only an authorized app should show sensitive data.

After the `-appConnectIsReady:` callback method is called, check the value of the `authState` property. Do the following:

- Remove the indication that the app is initializing.
- If the status is not `ACAUTHSTATE_AUTHORIZED`, do not allow the user to see or access sensitive data.
- If the status is not `ACAUTHSTATE_AUTHORIZED`, display the `authMessage` string.

- If the status is `ACAUTHSTATE_AUTHORIZED`, allow the user to see and access sensitive data. Typically, the app does not display the `authMessage` string when the status is `ACAUTHSTATE_AUTHORIZED`.

On any updates to authorization status while the app is running, the AppConnect library updates the properties, and then calls the `-appConnect:authStateChangedTo:withMessage:` method.

Authorization methods

Your app uses the following methods to receive updates to the authorization status and report how the app handled the updates.

- The -appConnect:authStateChangedTo:withMessage: callback method
- The -authStateApplied:message: acknowledgment method
- The -displayMessage: method

The -appConnect:authStateChangedTo:withMessage: callback method

You are required to implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *) appConnect
           authStateChangedTo:(ACAuthState)newAuthState
           withMessage:(NSString *)message;
```

When a change has occurred to the user's authorization status, the AppConnect library:

1. Sets the `authState` property on the AppConnect object to the new `ACAuthState` value.
2. Sets the `authMessage` property on the AppConnect object to a string explaining the new authorization status.
3. Calls the `-appConnect:authStateChangedTo:withMessage:` method.

The method provides the following in its parameters:

- the new authorization status as a `ACAuthState` value
- an `NSString`, which is a message explaining the new authorization status

Your app then handles the new status as follows:

TABLE 10. AUTHORIZATION STATUS HANDLING

New status	App actions
`ACAUTHSTATE_UNAUTHORIZED`	• Exits any sensitive part of the application. • Stops allowing the user to access sensitive data and views. • Displays the message received in the callback method that explains the authorization status change. • Calls the `-authStateApplied:message:` method.
`ACAUTHSTATE_AUTHORIZED`	• Allows the user to access sensitive data and views. • Calls the `-authStateApplied:message:` method.
`ACAUTHSTATE_RETIRED`	• Exits any sensitive part of the application. • Deletes all sensitive data, including any stored authentication credentials, data in files, keychain items, pasteboard data, and any other persistent

TABLE 10. AUTHORIZATION STATUS HANDLING (CONT.)

New status	App actions
	storage. • Displays the message received in the callback method that explains the authorization status change. • Calls the `-authStateApplied:message:` method.

NOTE: The AppConnect library can call the callback method when *only* the explanatory string, but not the authorization status, has changed. When the status is `ACAUTHSTATE_UNAUTHORIZED` or `ACAUTHSTATE_RETIRED`, the message typically contains a new reason for the status. Display the new message.

The -authStateApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void)authStateApplied:(ACPolicyState)policyState message:(NSString *)message;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the new authorization status. Pass the value `ACPOLICY_APPLIED` if the app successfully handled the new status. Otherwise, pass the value `ACPOLICY_ERROR`. Passing the value `ACPOLICY_UNSUPPORTED` is not allowed, because every app must handle authorization status changes.
- an `NSString` explaining the `ACPolicyState` value.
  Typically, you use this string to report the reason the app failed to apply the new authorization status. The string is reported in the MobileIron server log files.

The -displayMessage: method

The following method on the AppConnect singleton causes the MobileIron client app to display the current authorization status message:

```
-(void)displayMessage:(NSString *)message withCompletion:(void(^)(BOOL success))completion;
```

In most cases, your production app does not use this method. Your production app is responsible for displaying the message that it receives in the notification method for an authorization status change. Your app controls exactly when and how to display the string.

However, you can temporarily use this method when your app is under development. For example, when the status changes to `ACAUTHSTATE_UNAUTHORIZED`, your app must exit all sensitive views. This requirement can make displaying the message difficult, depending on the application. In this case, use the `-displayMessage:` method until you are able to fully develop your app.

App-specific configuration API details

The AppConnect for iOS API provides properties and methods that allow an app to receive app-specific configuration from the MobileIron server. For an overview of this feature, see Configuration specific to the app.

The config property

The read-only `config` property on the AppConnect singleton is an `NSDictionary` object. It contains the current key-value pairs for the app-specific configuration.

Whenever changes to the key-value pairs occur, the AppConnect library:

1. updates the `config` property
2. calls the `-appConnect:configChangedTo:` method to provide your app the current configuration.

When your app launches:

- Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.
- Wait for the `-appConnectIsReady:` callback method before accessing the `config` property.
- After the `-appConnectIsReady:` callback method is called, check the value of the `config` property. It contains the key-value pairs, if any, that are configured on the MobileIron server for the app. If no key-value pairs are configured, the `config` property is an `NSDictionary` object with no entries.
- Apply the configuration according to your application's requirements and logic.

App-specific configuration methods

Your app uses the following methods to receive app-specific configuration updates and report how the app handled the updates.

The -appConnect:configChangedTo: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect configChangedTo:(NSDictionary *)newConfig;
```

Implement this method only if your app uses app-specific configuration key-value pairs that the MobileIron server administrator configures on the server Admin Portal.

When a change has occurred to the app-specific configuration on the MobileIron server, the AppConnect library:

1. Sets the `config` property on the AppConnect object to the new `NSDictionary` value.
2. Calls the `-appConnect:configChangedTo:` method, which provides the new `NSDictionary` value in its parameter.

Your app then:

- applies the new configuration according to your application's requirements and logic.
- calls the `-configApplied:message:` method.

The -configApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void)configApplied:(ACPolicyState)policyState message:(NSString *)message;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the app-specific configuration updates.
  Pass the value `ACPOLICY_APPLIED` if the app successfully handled the updates. Otherwise, pass the value `ACPOLICY_ERROR`. Pass the value `ACPOLICY_UNSUPPORTED` if your app does not support app-specific configuration. If you do not implement the `-configApplied:message` method, the AppConnect singleton behaves as if you passed it `ACPOLICY_UNSUPPORTED`.
- an `NSString` explaining the `ACPolicyState` value.
  Typically, you use this string to report the reason the app failed to apply the app-specific configuration updates. The string is reported in the MobileIron server log files.

Pasteboard policy API details

The AppConnect for iOS API provides properties and methods that allow an app to handle its pasteboard policy as determined by the MobileIron server. For an overview of this feature, see Data loss prevention policies.

This policy determines whether your app is allowed to copy content *to* the pasteboard. This policy does not impact whether your app is allowed to paste content *from* the pasteboard into your app.

The ACPasteboardPolicy enumeration

The `ACPasteboardPolicy` enumeration provides the possible pasteboard statuses for the app:

```
typedef enum {
   ACPASTEBOARDPOLICY_UNAUTHORIZED =  0, // The application cannot write data
                                         // to the pasteboard.
                                         // The AppConnect library enforces this status
                                         // and ensures that the app cannot modify the
                                         // pasteboard contents.

    ACPASTEBOARDPOLICY_AUTHORIZED   =  1, // The application may write data to the pasteboard
                                          // which gets shared among all apps.
                                          // (Both AppConnect and non-AppConnect apps
                                          // can read this data).

    ACPASTEBOARDPOLICY_SECURECOPY    =  2 // The application may write data to the general
                                          // pasteboard which is shared with authorized
                                          // AppConnect apps.
                                          // The AppConnect library implements the underlying
                                          // technology so that the data written to the
                                          // general pasteboard by one AppConnect app is only
```

```
                                                // readable by authorized AppConnect apps.
} ACPasteboardPolicy;
```

Handle the pasteboard policy status as follows:

- Both `ACPASTEBOARDPOLICY_AUTHORIZED` and `ACPASTEBOARDPOLICY_SECURECOPY` indicate that copying content to the pasteboard is allowed. The AppConnect library handles making sure all apps or only AppConnect apps can paste the data. When the value is `ACPASTEBOARDPOLICY_SECURECOPY`, the AppConnect library encrypts the data copied to the pasteboard, and decrypts the data when it is pasted to another AppConnect app.
- The status `ACPASTEBOARDPOLICY_UNAUTHORIZED` indicates that writing data to the pasteboard is not allowed. The AppConnect library enforces the status `ACPASTEBOARDPOLICY_UNAUTHORIZED`. Therefore, with this status, *even if you use an API to write to the pasteboard, the data is not written*.
  Exceptions to this rule exist. For some iOS APIs, such as QLPreviewController, it is not possible to prevent data from being written to the pasteboard. If your app uses such APIs, when the status is `ACPASTEBOARDPOLICY_UNAUTHORIZED`, change your app's behavior so that it does not use that API. However, for some apps, changing the behavior is not possible, due to, for example, an unacceptable degradation in the app's capabilities. In that case, your app should indicate that it does not support the pasteboard policy, as described in The -pasteboardPolicyApplied:message: acknowledgment method.

Although the AppConnect library does not allow writing data to the pasteboard, your app should disable special user interfaces, if any, that it uses for copying content to the pasteboard. By disabling such user interfaces, your app does not give the end user the impression that copying is possible when the AppConnect library has disabled it.

Impact on the pasteboard policy of secure services availability

The pasteboard policy `ACPASTEBOARDPOLICY_SECURECOPY` depends on secure services being available. If secure services are not available and the pasteboard policy is `ACPASTEBOARDPOLICY_SECURECOPY`:

- Writing content to the pasteboard (copying) fails. No data is written.
- Reading content from the pasteboard (pasting) reads unsecured content, if any.

The pasteboardPolicy property

The read-only `pasteboardPolicy` property on the AppConnect singleton contains an `ACPasteboardPolicy` value. The value reflects the current status of the pasteboard policy for the app.

The AppConnect library enables or disables the app's ability to copy content to the pasteboard depending on the `pasteboardPolicy` value:

- Copying is enabled for `ACPASTEBOARDPOLICY_AUTHORIZED`.
- Copying is disabled for `ACPASTEBOARDPOLICY_UNAUTHORIZED`.
- Copying is enabled for `ACPASTEBOARDPOLICY_SECURECOPY` if the `secureServicesAvailability` property has the value `ACSECURESERVICESAVAILABILITY_AVAILABLE`.
- Copying is disabled for `ACPASTEBOARDPOLICY_SECURECOPY` if the `secureServicesAvailability` property has the value `ACSECURESERVICESAVAILABILITY_UNAVAILABLE`.

When your app launches:

1. Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.
2. Wait for the `-appConnectIsReady:` callback method before accessing the `pasteboardPolicy` property.
3. After the `-appConnectIsReady:` callback method is called, depending on the `pasteboardPolicy` value, disable or enable special user interfaces, if any, that the app uses for copying content to the pasteboard. Although the AppConnect library enables or disables writing data to the pasteboard, your app should not give the end user the impression that copying is possible when the AppConnect library has disabled it.

Whenever the policy changes, the AppConnect library:

1. updates the `pasteboardPolicy` property.
2. calls the `-appConnect:pasteboardPolicyChangedTo:` method to provide your app the current pasteboard policy.

Pasteboard policy methods

Your app uses the following methods to receive pasteboard policy updates and to report how the app handled the updates.

- The -appConnect:pasteboardPolicyChangedTo: callback method
- The -pasteboardPolicyApplied:message: acknowledgment method
- The -appConnect:copyAttemptedWhenUnauthorized: callback method

The -appConnect:pasteboardPolicyChangedTo: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect pasteboardPolicyChangedTo:
                                  (ACPasteboardPolicy)newPasteboardPolicy;
```

Implement this method only if your app copies content *to* the pasteboard. This policy does not impact whether your app is allowed to paste content *from* the pasteboard into your app.

When a change has occurred to the pasteboard policy on the MobileIron server, the AppConnect library:

1. Sets the `pasteboardPolicy` property on the AppConnect object to the new `ACPasteboardPolicy` value.
2. Disables or enables copying to the pasteboard as follows:
   - Enables copying for `ACPASTEBOARDPOLICY_AUTHORIZED`.
   - Disables copying for `ACPASTEBOARDPOLICY_UNAUTHORIZED`.
   - Enables copying for `ACPASTEBOARDPOLICY_SECURECOPY` if the `secureServicesAvailability` property has the value `ACSECURESERVICESAVAILABILITY_AVAILABLE`.
   - Disables copying for `ACPASTEBOARDPOLICY_SECURECOPY` if the `secureServicesAvailability` property has the value `ACSECURESERVICESAVAILABILITY_UNAVAILABLE`.
3. Calls the `appConnect:pasteboardPolicyChangedTo:` method, which provides the new `ACPasteboardPolicy` value in its parameter.

Your app then:

- Disables or enables special user interfaces, if any, that the app uses for copying content to the pasteboard. Although the AppConnect library enables or disables writing data to the pasteboard, your app should not give the end user the impression that copying is possible when the AppConnect library has disabled it.
- calls the `-pasteboardPolicyApplied:message:` method.

The -pasteboardPolicyApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void)pasteboardPolicyApplied:(ACPolicyState)policyState message:(NSString *)message;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the pasteboard policy update. Pass the value `ACPOLICY_APPLIED` if the app successfully handled the update. Otherwise, pass the value `ACPOLICY_ERROR`. Pass the value `ACPOLICY_UNSUPPORTED` if your app does not support copying content to the pasteboard. If you do not implement the `-pasteboardPolicyApplied:message` method, the AppConnect singleton behaves as if you passed it `ACPOLICY_UNSUPPORTED`.
- an `NSString` explaining the `ACPolicyState` value.
  Typically, you use this string to report the reason the app failed to apply the pasteboard policy update. The string is reported in the MobileIron server log files.

The -appConnect:copyAttemptedWhenUnauthorized: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect copyAttemptedWhenUnauthorized:
                                  (ACPasteboardPolicy)pasteboardPolicy;
```

This method is useful because when the pasteboard policy is `ACPASTEBOARDPOLICY_UNAUTHORIZED`, iOS behavior still causes the copy button to display. An end user who taps the copy button sometimes expects that text has been copied. You can implement this method to alert the end user that no text has been copied.

For example:

```
-(void) appConnect:(AppConnect *)appConnect copyAttemptedWhenUnauthorized:
                                                (ACPasteboardPolicy)pasteboardPolicy {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Copy not allowed"
                          message:@"You are not allowed to copy from this app."
                          delegate:nil cancelButtonTitle:@"OK" otherButtonTitles: nil];
    [alert show];
}
```

Drag and drop policy API details

The drag and drop policy on the MobileIron server specifies whether AppConnect apps can drag content to all other apps, to only other AppConnect apps, or not at all.

The AppConnect library enforces this policy. When the policy allows dragging content to only other AppConnect apps, the AppConnect library notifies your app when the device user attempts to drag content to a non-AppConnect app. Your app can then notify the device user of the situation.

Drag and drop policy method

The AppConnect library enforces the MobileIron server's drag and drop policy. Therefore, your app is not aware of the policy setting. Therefore, no enumeration describing the settings is necessary, and you do not have to handle the setting or changes to the setting in your app.

However, when the policy allows dragging content to only other AppConnect apps, a device user can still attempt to drag content to non-AppConnect apps. That attempt fails. In this situation, the AppConnect library calls a callback method.

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
- (void)appConnectAttemptedDragAndDropToNonAppConnectApp:(AppConnect *)appConnect;
```

You can implement this method to alert the end user that dragging content to non-AppConnect apps is not allowed.

For example:

```
-(void) appConnect:(AppConnect *)appConnect appConnectAttemptedDragAndDropToNonAppConnectApp:
{

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Drag and drop not allowed"
                message:@"You are not allowed to drag content from this app to unsecured apps."
                delegate:nil cancelButtonTitle:@"OK" otherButtonTitles: nil];
    [alert show];
}
```

Open In policy API details

The AppConnect for iOS API provides properties and methods that allow an app to handle its Open In policy as determined by the MobileIron server. For an overview of this feature, see Data loss prevention policies.

Specifically, when an app is allowed to use Open In, it can share a document with another app, or an app's extension, or the native iOS mail app. This capability:

- is usually presented to the user as an Open In menu item.
- includes sending documents or document portions by encoding them in custom URLs handled by other applications.

Internally, an app uses the UIDocumentInteractionController, QLPreviewController (which in turn uses UIDocumentInteractionController), and UIActivityViewController classes. The class which the app uses impacts Open In handling as described in Overview of Open In handling.

Overview of Open In handling

The behavior of the AppConnect library, and the actions your app takes, depend on the Open In policy status.

The possible status values are:

- `ACOPENINPOLICY_AUTHORIZED` -- The application is allowed to use Open In.
- `ACOPENINPOLICY_UNAUTHORIZED` -- The application is not allowed to use Open In.
- `ACOPENINPOLICY_WHITELIST` -- The application is allowed to use Open In to send documents only to apps in the whitelist. To put the iOS native email app in the whitelist, the whitelist must contain both of these bundle IDs: com.apple.UIKit.activity.Mail and com.apple.mobilemail.

IMPORTANT:

- Regardless of the Open In policy status, when an app makes an Open In request, iOS always displays all the apps that support the document type.
- Do not use UIActivityViewController to perform Open In functionality. Because of iOS implementation, the AppConnect library cannot determine which app the end user selects, and therefore, whether it is in the whitelist. To ensure security, the AppConnect library does not allow Open In to any app when the Open In policy is `ACOPENINPOLICY_WHITELIST` and the class used is UIActivityViewController.

The following table summarizes the behavior of the AppConnect library and the actions your app takes for each Open In status. It assumes you use UIDocumentInteractionController, and do not use UIActivityViewController.

TABLE 11. OPEN IN ACTIONS TAKEN BY THE APPCONNECT LIBRARY AND YOUR APP

Open In status	AppConnect library actions	Your app's actions
`AUTHORIZED`	The AppConnect library performs no actions on Open In behavior.	Enable user interfaces, if any, that give the user the option to use Open In. For example, if your app presents a menu item for Open In, the menu item should be enabled.
`UNAUTHORIZED`	If a user taps on any of the apps: • the AppConnect library substitutes a dummy file with a mangled name. Therefore, the target app cannot open the file. Target app error handling varies. For example, some apps display an error pop-up. • The AppConnect library also calls this callback method if your app implemented it: -appConnect:openInAttemptedWhenACOpenInPolicyBlocked:	• Disable user interfaces, if any, that give the user the option to use Open In. For example, if your app presents a menu item for Open In, the menu item should be disabled. By disabling such user interfaces, your app does not give the end user the impression that Open In is possible when the AppConnect library has disabled it. • Implement the callback method -appConnect: openInAttemptedWhenACOpenInPolicyBlocked: In the method, notify the user that Open In is not allowed. Note that if you disabled all Open In user interfaces, this method will not be called.
`WHITELIST`	If a user taps on an app that is not in the whitelist:	• Enable user interfaces, if any, that give the user the option to use Open In. For example,

TABLE 11. OPEN IN ACTIONS TAKEN BY THE APPCONNECT LIBRARY AND YOUR APP (CONT.)

Open In status	AppConnect library actions	Your app's actions
	• the AppConnect library substitutes a dummy file with a mangled name. Therefore, the target app cannot open the file. Target app error handling varies. For example, some apps display an error pop-up. • The AppConnect library also calls this callback method if your app implemented it: -appConnect:openInAttemptedWhenACOpenInPolicyBlocked:	if your app presents a menu item for Open In, the menu item should be enabled. • Implement the callback method -appConnect: openInAttemptedWhenACOpenInPolicyBlocked: In the method, notify the user that Open In is not allowed to the selected app.

The ACOpenInPolicy enumeration

The `ACOpenInPolicy` enumeration provides the possible Open In statuses for the app:

```
typedef enum {
    ACOPENINPOLICY_UNAUTHORIZED =  0, // The application may not use Open In.
    ACOPENINPOLICY_AUTHORIZED   =  1, // The application may use Open In.
    ACOPENINPOLICY_WHITELIST    =  2, // The application may only use Open In to send
                                      // documents to applications in the whitelist.
} ACOpenInPolicy;
```

The openInPolicy and openInWhitelist properties

The following read-only properties on the AppConnect singleton relate to the Open In policy:

TABLE 12. OPEN IN PROPERTIES ON THE APPCONNECT SINGLETON

Property	Description
`openInPolicy`	An `ACOpenInPolicy` value that indicates the current status of the Open In policy for the app.
`openInWhitelist`	An `NSSet` object that contains `NSString` objects. Each string is the bundle ID of an app in the whitelist. The whitelist is the set of apps to which your app is allowed to send documents. Because the AppConnect library enforces Open In to only the whitelisted apps, your app uses this list only if it wants to inform the user about the list. NOTE: When the Open In policy on the MobileIron server specifies "All AppConnect apps", the Open In status value is `ACOPENINPOLICY_WHITELIST`. The `openInWhitelist` lists all the currently authorized AppConnect apps. Therefore, your app handles the "All AppConnect apps" server setting the same way it handles the "whitelist" server setting.

When your app launches:

- Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.

- Wait for the `-appConnectIsReady:` callback method before accessing the `openInPolicy` and `openInWhitelist` properties.

After the `-appConnectIsReady:` callback method is called, enable or disable user interfaces, if any, that give the user the option to use the Open In feature, depending on the `openInPolicy` property value.

Whenever changes to the Open In policy or whitelist occur, the AppConnect library:

1. Updates the properties.
2. Calls the `-openInPolicyChangedTo:whitelist:` method to provide your app the current information.

Open In policy methods

Your app uses the following methods to receive Open In policy updates and to report how the app handled the updates.

The -appConnect:openInPolicyChangedTo:whitelist: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect openInPolicyChangedTo:
          (ACOpenInPolicy)newOpenInPolicy whitelist:(NSSet *)newWhitelist;
```

Implement this method only if your app uses the Open In feature.

When a change has occurred to the Open In policy on the MobileIron server, the AppConnect library:

1. Sets the `openInPolicy` and `openInWhitelist` properties on the AppConnect object to the new values.
2. Calls the `-appConnect:openInPolicyChangedTo:whitelist` method, which provides the new values in its parameters.

Your app then:

- Enable or disable user interfaces, if any, that give the user the option to use the Open In feature, depending on the `openInPolicy` property value.
- calls the `-appConnect:openInPolicyApplied:message:` method.

The -openInPolicyApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void)openInPolicyApplied:(ACPolicyState)policyState message:(NSString *)message;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the Open In policy update.

Pass the value `ACPOLICY_APPLIED` if the app successfully handled the update. Otherwise, pass the value `ACPOLICY_ERROR`. Pass the value `ACPOLICY_UNSUPPORTED` if your app does not support the Open In feature. If you do not implement the `-openInPolicyApplied:message` method, the AppConnect singleton behaves as if you passed it `ACPOLICY_UNSUPPORTED`.

- an `NSString` explaining the `ACPolicyState` value.
  Typically, you use this string to report the reason the app failed to apply the Open In policy update. The string is reported in the MobileIron server log files.

The -appConnect:openInAttemptedWhenACOpenInPolicyBlocked: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect openInAttemptedWhenACOpenInPolicyBlocked:
                                    (ACOpenInPolicy)OpenInPolicy;
```

This method is useful because even when the Open In policy is `ACOPENINPOLICY_UNAUTHORIZED` or `ACOPENINPOLICY_WHITELIST`, when an app makes an Open In request, iOS still displays all apps that support the document type. An end user who taps an app sometimes expects the Open In operation to be successful. You can implement this method to alert the end user that Open In is not allowed for the selected app.

Note that this method is also called for Open In requests to the iOS native email app when the policy is `ACOPENINPOLICY_UNAUTHORIZED` or `ACOPENINPOLICY_WHITELIST` and the native email app is not on the whitelist. For example, with these policy settings, this method is called when the device user taps a :mailTo link in a UIWebView, WKWebView, or UITextView, or when the app attempts to display a MFMailComposeViewController.

For example:

```
-(void) appConnect:(AppConnect *)appConnect
          openInAttemptedWhenACOpenInPolicyBlocked: (ACOpenInPolicy)OpenInPolicy {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Open In not allowed"
                          message:@"You are not allowed to share a document with this app."
                          delegate:nil cancelButtonTitle:@"OK" otherButtonTitles: nil];
    [alert show];
}
```

The -appConnect:openURLAttemptedWhenUnauthorizedForURL: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect openURLAttemptedWhenUnauthorizedForURL:
                                    (NSURL *)openURL;
```

This method is called when your app has called `-openURL:` with the `mailto` scheme and either:

- the Open In policy is `ACOPENINPOLICY_UNAUTHORIZED`
- the Open In policy is `ACOPENINPOLICY_WHITELIST`, and the whitelist does not contain the bundle ID of an app that can handle the URL, such as the native iOS email app.
  However, if the AppConnect app Email+ is installed on the device, it is opened and the callback method is not called.

Use this method to notify the end user that opening an app to handle the request is not allowed due to the Open In policy.

Info.plist key related to the Open In policy

Your app can override the Open In policy when the policy blocks the iOS native email app when the app calls `openURL:` with the `mailto:` scheme. Overriding the Open In policy for this scenario means that the iOS native email app is opened even though the Open In policy is one of the following:

- `ACOPENINPOLICY_UNAUTHORIZED`
- `ACOPENINPOLICY_WHITELIST`, and the whitelist does not contain the bundle IDs of the native iOS email app.

To override the Open In policy for this scenario, add the key MI_AC_DISABLE_SCHEME_BLOCKING with the value `YES` in the MI_APP_CONNECT dictionary in the app's Info.plist.

NOTE: The MobileIron server administrator can also override the Open In policy for this scenario by adding the key MI_AC_DISABLE_SCHEME_BLOCKING with the value `true` to the app's app-specific configuration.

Open From policy API details

IMPORTANT: T **Open From does not work on iOS 13 devices.**

The AppConnect for iOS API provides properties and methods that allow an app to handle its Open From policy as determined by the MobileIron server. For an overview of this feature, see Data loss prevention policies.

Specifically, when an app is allowed to use Open From, it can receive a document shared from another app (or another app's extension) that uses the Open In iOS feature.

- Overview of Open From handling
- The ACOpenFromPolicy enumeration
- The openFromPolicy and openFromWhitelist properties
- Open From policy methods
- Open From policy API details

Overview of Open From handling

The behavior of the AppConnect library, and the actions your app takes, depend on the Open From policy status.

The possible status values are:

- `ACOPENFROMPOLICY_AUTHORIZED` -- The app is allowed to receive documents shared by any app that uses Open In.
- `ACOPENFROMPOLICY_UNAUTHORIZED` -- The app is not allowed to receive documents shared by any app that uses Open In.
- `ACOPENFROMPOLICY_WHITELIST` -- The app is allowed to receive documents shared by another app using Open In only if the other app is in the whitelist. To put the iOS native email app in the whitelist, the whitelist must contain both of these bundle IDs: com.apple.UIKit.activity.Mail and com.apple.mobilemail.

IMPORTANT:

When an app makes an Open In request, iOS always displays all the apps that support the document type, regardless of:

- the requesting app's Open In policy status if it is an AppConnect app
- the receiving app's Open From policy status if it is an AppConnect app

Although the AppConnect library enforces the Open From policy, this iOS behavior means that your app might want to keep the user informed of failed attempts. The following table summarizes the behavior of the AppConnect library and recommended actions for your app relating to Open From.

TABLE 13. OPEN FROM ACTIONS TAKEN BY THE APPCONNECT LIBRARY AND YOUR APP

Open In status	AppConnect library actions	Your app's actions
`AUTHORIZED`	The AppConnect library performs no actions on Open From behavior.	None
`UNAUTHORIZED`	The AppConnect library does not allow another app to Open In to your app. Additionally, if a user chooses to Open In to your app, the AppConnect library calls this callback method if your app implemented it. -appConnect:openFromAttemptedWhenACOpenFromPolicyBlocked:	Implement the callback method. In the method, notify the user that using Open In from the specified app to your app is not allowed.
`WHITELIST`	The AppConnect library does not allow an app that is not on the whitelist to Open In to your app. Additionally, if a user chooses to Open In to your app from an app that is not in the whitelist, the AppConnect library calls this callback method if your app implemented it: -appConnect:openFromAttemptedWhenACOpenFromPolicyBlocked:	Implement the callback method. In the method, notify the user that Open From the specified app is not allowed.

The ACOpenFromPolicy enumeration

The `ACOpenFromPolicy` enumeration provides the possible Open From statuses for the app:

```
typedef enum {
    ACOPENFROMPOLICY_UNAUTHORIZED =  0, // The app is allowed to receive documents shared by
                                        // any app that uses Open In.
    ACOPENFROMPOLICY_AUTHORIZED   =  1, // The app is not allowed to receive documents shared
                                        // by any app that uses Open In.
    ACOPENFROMPOLICY_WHITELIST    =  2, // The app is allowed to receive documents shared by
                                        // another app using Open In only if the other app
                                        // is in the whitelist.

} ACOpenFromPolicy;
```

The openFromPolicy and openFromWhitelist properties

The following read-only properties on the AppConnect singleton relate to the Open From policy:

TABLE 14. OPEN FROM PROPERTIES ON THE APPCONNECT SINGLETON

Property	Description
`openFromPolicy`	An `ACOpenFromPolicy` value that indicates the current status of the Open From policy for the app.
`openFromWhitelist`	An `NSSet` object that contains `NSString` objects. Each string is the bundle ID of an app in the whitelist. The whitelist is the set of apps from which your app is allowed to receive documents. Because the AppConnect library enforces Open From from only the whitelisted apps, your app uses this list only if it wants to inform the user about the list. NOTE: When the Open From policy on the MobileIron server specifies "All AppConnect apps", the Open From status value is `ACOPENFROMPOLICY_WHITELIST`. The `openFromWhitelist` lists all the currently authorized AppConnect apps. Therefore, your app handles the "All AppConnect apps" server setting the same way it handles the "whitelist" server setting.

When your app launches:

- Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.
- Wait for the `-appConnectIsReady:` callback method before accessing the `openFromPolicy` and `openFromWhitelist` properties.

Whenever changes to the Open From policy or whitelist occur, the AppConnect library:

1. Updates the properties.
2. Calls the `-openFromPolicyChangedTo:whitelist:` method to provide your app the current information.

Open From policy methods

Your app uses the following methods to receive Open From policy updates and to report how the app handled the updates.

- The -appConnect:openFromPolicyChangedTo:whitelist: callback method
- The -openFromPolicyApplied:message: acknowledgment method
- The -appConnect:openFromAttemptedWhenACOpenFromPolicyBlocked: callback method

The -appConnect:openFromPolicyChangedTo:whitelist: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect openFromPolicyChangedTo:
          (ACOpenInPolicy)newOpenFromPolicy whitelist:(NSSet<NSString *>)newWhitelist;
```

Implement this method only if:

- your app handles documents from other apps using Open In, and
- your app uses the Open From policy value or whitelist in some way, such as displaying information about it to the user. The AppConnect library enforces the policy.

When a change has occurred to the Open From policy on the MobileIron server, the AppConnect library:

1. Sets the `openFromPolicy` and `openFromWhitelist` properties on the AppConnect object to the new values.
2. Calls the `-appConnect:openFromPolicyChangedTo:whitelist` method, which provides the new values in its parameters.

Your app then:

- Can access the new values. The app can use them, for example, to inform the end user about allowed Open From apps.
- calls the `-appConnect:openFromPolicyApplied:message:` method.

The -openFromPolicyApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void)openFromPolicyApplied:(ACPolicyState)policyState message:(NSString *)message;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the Open From policy update. Pass the value `ACPOLICY_APPLIED` if the app successfully handled the update. Otherwise, pass the value `ACPOLICY_ERROR`. Pass the value `ACPOLICY_UNSUPPORTED` if your app does not support the Open From feature. If you do not implement the `-openInPolicyApplied:message` method, the AppConnect singleton behaves as if you passed it `ACPOLICY_UNSUPPORTED`.
- an `NSString` explaining the `ACPolicyState` value.
  Typically, you use this string to report the reason the app failed to apply the Open From policy update. The string is reported in the MobileIron server log files.

The -appConnect:openFromAttemptedWhenACOpenFromPolicyBlocked: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect openFromAttemptedWhenACOpenFromPolicyBlocked:
                                  (ACOpenFromPolicy)OpenFromPolicy
                                  sourceApplication:(NSString *)sourceApplicationId;
```

This method is useful because even when the Open From policy is `ACOPENFROMPOLICY_UNAUTHORIZED` or `ACOPENFROMPOLICY_WHITELIST`, when another app makes an Open In request, iOS still displays all apps that support the document type. An end user who taps an app expects the Open In operation to be successful. You can implement this method to alert the end user that your app is not allowed to receive documents from the other app.

Print policy API details

The AppConnect for iOS API provides properties and methods that allow an app to handle its print policy as determined by the MobileIron server. For an overview of this feature, see Data loss prevention policies.

The ACPrintPolicy enumeration

The `ACPrintPolicy` enumeration provides the possible print statuses for the app:

```
typedef enum {
    ACPRINTPOLICY_UNAUTHORIZED =  0, // The application may not use Print.
    ACPRINTPOLICY_AUTHORIZED   =  1, // The application may use Print.
} ACPrintPolicy;
```

The printPolicy property

The read-only `printPolicy` property on the AppConnect singleton contains an `ACPrintPolicy` value. The value reflects the current status of the print policy for the app.

When your app launches:

- Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.
- Wait for the `-appConnectIsReady:` callback method before accessing the `printPolicy` property.
- After the `-appConnectIsReady:` callback method is called, enable or disable the app's ability to print depending on the `printPolicy` property value.

Whenever the policy changes, the AppConnect library:

1. updates the `printPolicy` property.
2. calls the `-appConnect:printPolicyChangedTo:` method to provide your app the current print policy.

Print policy methods

Your app uses the following methods to receive print policy updates and to report how the app handled the updates.

The -appConnect:printPolicyChangedTo: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect printPolicyChangedTo:
                                  (ACPrintPolicy)newPrintPolicy;
```

Implement this method only if your app is able to print.

When a change has occurred to the print policy on the MobileIron server, the AppConnect library:

1. Sets the `printPolicy` property on the AppConnect object to the new `ACPrintPolicy` value.
2. Calls the `-appConnect:printPolicyChangedTo:` method, which provides the new `ACPrintPolicy` value in its parameter.

Your app then:

- Enables or disables its ability to print depending on the passed `ACPrintPolicy` value.
- calls the `-printPolicyApplied:message:` method.

The -printPolicyApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void) appConnect:(AppConnect *)appConnect printPolicyChangedTo:
                                  (ACPrintPolicy)newPrintPolicy;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the print policy update.
  Pass the value `ACPOLICY_APPLIED` if the app successfully handled the update. Otherwise, pass the value `ACPOLICY_ERROR`. Pass the value `ACPOLICY_UNSUPPORTED` if your app does not support print. If you do not implement the `-printPolicyApplied:message` method, the AppConnect singleton behaves as if you passed it `ACPOLICY_UNSUPPORTED`.
- an `NSString` explaining the `ACPolicyState` value.
  Typically, you use this string to report the reason the app failed to apply the print policy update. The string is reported in the MobileIron server log files.

Log messages API details

The AppConnect for iOS API provides properties and methods that allow an app to log messages at various severity levels to the device's console or to files. For an overview of this feature, see Log messages.

The ACLogLevel enumeration

The `ACLogLevel` enumeration provides the possible log levels:

```
typedef enum {
    ACLOGLEVEL_ERROR   = 0, // Error messages
    ACLOGLEVEL_WARNING = 1, // Warning messages
    ACLOGLEVEL_STATUS  = 2, // Significant status messages such as app launch
                            // and major user actions.
    ACLOGLEVEL_INFO    = 3, // Additional informational messages
    ACLOGLEVEL_VERBOSE = 4, // Verbose messages which may include sensitive information
    ACLOGLEVEL_DEBUG   = 5, // Debug messages which may include sensitive information
} ACLogLevel;
```

Log level descriptions and examples

The following table provides guidelines about when to use each log level:

TABLE 15. LOG LEVEL DESCRIPTIONS

Log level	Description	May contain sensitive data?	Examples of when to use
`ACLOGLEVEL_ERROR`	For events that block access to part or all of the app.	No	• Uncaught exception • Corrupt or missing data • Network timeout • Digital signature validation error • Certificate error • Failed assertion • Exhausted retries • Error that is reported to the user
`ACLOGLEVEL_WARNING`	For events that are suspicious, but not quite failures like errors.	No	• Caught exception that is ignored • Failed login due to bad user credentials • Unexpected data that is ignored • Network connection that is established just before timing out • Retrying • Attempted forward compatibility. For example, the app was developed with and tested against version 1 of the server, but the server reported version 2. • Feature disabled due to low battery • User attempted something that is not currently allowed or available • Warning that is reported to the user
`ACLOGLEVEL_STATUS`	For major changes in the state of the app	No	• App launched • Version information • Successfully logged in • Successfully opened, saved or closed a user document • Successfully deleted sensitive data when authorization state changed to `ACAUTHSTATE_RETIRED` • Notification received from the server • Status that is reported to the user
`ACLOGLEVEL_INFO`	For minor changes in the state of the app	No	• Changed views within the app • Heartbeat sent to server • App entered foreground or background, became active or inactive, and other UIApplicationDelegate app state changes • Interaction with device hardware, such

TABLE 15. LOG LEVEL DESCRIPTIONS (CONT.)

Log level	Description	May contain sensitive data?	Examples of when to use
			as taking a photo • Changes to non-sensitive user preferences
`ACLOGLEVEL_VERBOSE`	For more extensive information, possibly including sensitive details	Yes	• Server addresses of requests and resulting HTTP status codes • Sensitive details of messages of less verbose log levels, such as the name of a saved file • Device identifiers
`ACLOGLEVEL_DEBUG`	For the most information, possibly including sensitive details	Yes	• Very precise user actions, such as touch events and keystrokes • URL request details • Memory and performance profiling information

Sensitive data examples

Include sensitive data only in messages logged at the verbose or debug levels. Examples of sensitive data are:

- User data, including document contents, document names, contact lists, notes, bookmarks
- Initial bytes of symmetric encryption keys, private encryption keys, passwords, certificates, signing identities, and cookies.

Only log initial bytes of these security-related values to ensure the values remain secure.

- Complete URLs and POST data
- Anything that may reveal the content of encrypted data, such as detailed error messages generated by parsing decrypted data

The logLevel property

```
+(ACLogLevel)logLevel;
```

The read-only `logLevel` class property on the AppConnect class contains an `ACLogLevel` value. The value reflects the current log level.

Use the following to get the `logLevel` value:

```
[AppConnect logLevel]
```

When your app calls the methods `+logAtLevel:format:`, `+logAtLevel:format:args:`, or `+logAtLevel:message:`, the AppConnect library logs a message only if the log level you pass to the method is less than or equal to the `logLevel` property's current value.

Your app can use the `logLevel` property to determine what work to do before calling one of the logging methods. For example, if the app gathers and formats a lot of data when the log level is `ACLOGLEVEL_DEBUG`, it can skip that logic when the log level is less than `ACLOGLEVEL_DEBUG`.

Log level methods

Your app uses the following methods to receive log level updates and to log messages to the device's console.

The -appConnect:logLevelChangedTo: callback method

You optionally implement this method to receive log level updates. The method is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect logLevelChangedTo:
                                  (ACLogLevel)newLogLevel;
```

Implement this method only if your app logs messages using the methods `+logAtLevel:format:` or `+logAtLevel:format:args:`.

When a change has occurred to the log level on the MobileIron server, the AppConnect library:

1. Sets the `logLevel` property on the AppConnect object to the new `ACLogLevel` value.
2. Calls the `-appConnect:logLevelChangedTo:` method, which provides the new `ACLogLevel` value in its parameter.

Your app can use the notification to start or stop gathering additional data that the app uses in logging.

logAtLevel class methods

Use the following AppConnect class methods to log messages:

```
+(void)logAtLevel:(ACLogLevel)logLevel format:(NSString *)format,
                                       ... NS_FORMAT_FUNCTION(2,3);

+(void)logAtLevel:(ACLogLevel)logLevel format:(NSString *)format
                                       args:(va_list)args NS_FORMAT_FUNCTION(2,0);

+(void)logAtLevel:(ACLogLevel)logLevel message:(NSString *)message;
```

These class methods use the logLevel parameter to determine whether to log a message. If `logLevel` parameter is less than or equal to the `logLevel` property, the methods log a message.

The methods prepend the format or message string with one of the following strings depending on the `logLevel` parameter:

- `[Error]`
- `[Warning]`
- `[Status]`
- `[Info]`
- `[Verbose]`
- `[Debug]`

Then:

- `+logAtLevel:format:` calls `NSLog()`, passing along the parameters.
- `+logAtLevel:format:args:` calls `NSLogv()`, passing along the parameters.
- `+logAtLevel:message:` calls `NSLog()`, passing along the parameter.

The following table describes the parameters of these methods:

TABLE 16. DESCRIPTIONS OF LOG LEVEL METHODS' PARAMETERS

Parameter	Description
`logLevel`	The `ACLogLevel` value for the message. The AppConnect library logs the message only if the passed log level is less than or equal to the current value of the `logLevel` property.
`format`	A format string. This parameter is an NSString, and can include format specifiers that NSString formatting methods support.
`...`	A comma-separated list of arguments to substitute into the format string in `+logAtLevel:format:`.
`args`	A `va_list` argument in `+logAtLevel:format:args` Use `+logAtLevel:format:args` if you want to explicitly prepare a `va_list` argument that contains the list of arguments that you pass to the logging method. Preparing a `va_list` argument is useful when you want to wrap AppConnect logging functionality with your own function.
`message`	A string. This parameter is an NSString, useful if you do not require format specifiers.

-logAtLevel:format:args: example

```
-(void)myDebugLogWithFormat:(NSString *)format, ... NS_FORMAT_FUNCTION(1,2) {

    va_list args;
    va_start(args, format);
    [AppConnect logAtLevel:ACLOGLEVEL_DEBUG format:format args:args];
    va_end(args);
}

-(void)someMethod {

    NSString *foo = @"FooValue";
    int bar = 4;
    [self myDebugLogWithFormat:@"Foo: %@ Bar: %i", foo, bar];
}
```

Log level methods and dual mode apps

A dual mode app can call the `+logAtLevel:format:`, `+logAtLevel:format:args:`, and `+logAtLevel:message:` methods and get the `+(ACLogLevel)logLevel` property even after calling `-stop:` on the AppConnect singleton object. When the AppConnect library is stopped, the log level is always `ACLOGLEVEL_STATUS`.

The notification method `-appConnect:logLevelChangedTo:` is not called when the AppConnect library is stopped.

Secure services API details

The AppConnect for iOS API provides properties and methods that allow an app to:

- determine whether secure services are available.
- handle its secure file I/O policy.
- perform file operations on secure, encrypted files.
- determine whether secure copy to the pasteboard is available.

For an overview of this feature, see "Data Protection" in Securing and managing the app using the AppConnect library.

The ACSecureServicesAvailability enumeration

The `ACSecureServicesAvailability` enumeration provides the possible secure services availability statuses for the app:

```
typedef enum {
    ACSECURESERVICESAVAILABILITY_UNAVAILABLE = 0, // Secure services are
                                                  // currently unavailable.
    ACSECURESERVICESAVAILABILITY_AVAILABLE   = 1, // Secure services are currently available.
} ACSecureServicesAvailability;
```

For more information about these values, see The secureServicesAvailability and secureFileIOPolicy properties .

The ACSecureFileIOPolicy enumeration

The `ACSecureFileIOPolicy` enumeration provides the possible secure file I/O policy statuses for the app:

```
typedef enum {
    ACSECUREFILEIOPOLICY_OPTIONAL = 0, // The application may store sensitive files using iOS
                                       // filesytem APIs or AppConnect secure file I/O.
    ACSECUREFILEIOPOLICY_REQUIRED = 1, // The application must store sensitive files
                                       // using only AppConnect secure file I/O.
} ACSecureFileIOPolicy;
```

NOTE: This policy is not configurable by the MobileIron server administrator; The server always requires secure file I/O. For more information about these values, see The secureServicesAvailability and secureFileIOPolicy properties .

The secureServicesAvailability and secureFileIOPolicy properties

The following read-only properties on the AppConnect singleton relate to secure services:

TABLE 17. SECURE SERVICES PROPERTIES ON THE APPCONNECT SINGLETON

Property	Description
`secureServicesAvailability`	An `ACSecureServicesAvailability` value that indicates whether secure services are currently available. Secure services availability depends on the availability of the encryption key. The encryption key is available only when all of the following are true: • the device user has entered the AppConnect passcode, or the device passcode when no AppConnect passcode is required • the app is authorized • the AppConnect singleton is ready (the `ready` property is `YES`)
`secureFileIOPolicy`	An `ACSecureFileIOPolicy` that contains the current status of the secure file I/O policy for the app. This secure file I/O policy is not configurable by the MobileIron server administrator; The server always requires secure file I/O.

When your app launches:

- Get the singleton AppConnect object and call its `-startWithLaunchOptions:` method.
- Wait for the `-appConnectIsReady:` callback method before accessing the `secureFileIOPolicy` property.

After the `-appConnectIsReady:` callback method is called:

- if the `secureServicesAvailability` property has the value `ACSECURESERVICESAVAILABILITY_AVAILABLE`, use secure file I/O APIs.
- If the `secureServicesAvailability` property has the value `ACSECURESERVICESAVAILABILITY_UNAVAILABLE`, wait for the notification of it changing to `ACSECURESERVICESAVAILABILITY_AVAILABLE` before using secure file I/O. Secure file I/O APIs fail when secure services are not available. The notification method is `-appConnect:secureServicesAvailabilityChangedTo:`.
- disable or enable special user interfaces for copying to the pasteboard if the pasteboard policy is `ACPASTEBOARDPOLICY_SECURECOPY`. Copying to the pasteboard fails if secure services are unavailable and the pasteboard policy is `ACPASTEBOARDPOLICY_SECURECOPY`.

NOTE: Because the secure file I/O policy is always set to required on the MobileIron server, the value of the `secureFileIOPolicy` property is always `ACSECUREFILEIOPOLICY_REQUIRED`. When secure file I/O is required, your app should use secure file I/O APIs (but only while secure services are available).

If the app becomes unauthorized, secure services change to unavailable. Similarly, if the AppConnect passcode auto-lock timeout expires, secure services also change to unavailable. When these situations occur, the AppConnect library calls the `-appConnect:secureServicesAvailabilityChangedTo:` notification method.

NOTE: Consider the case when the AppConnect passcode auto-lock timeout expires while your app is performing a file operation. The operations will fail. Make sure your app can handle the returned error conditions.

Secure services methods

Your app uses the following methods to receive secure services availability updates and to report how the app handled the updates.

- The -appConnect:secureServicesAvailabilityChangedTo: callback method
- The -appConnect:secureFileIOPolicyChangedTo: callback method
- The -secureFileIOPolicyApplied:message: acknowledgment method

The -appConnect:secureServicesAvailabilityChangedTo: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect secureServicesAvailabilityChangedTo:
           (ACSecureServicesAvailability)secureServicesAvailability;
```

Implement this method only if your app uses secure services.

When secure services availability changes, the AppConnect library:

1. Updates the `secureServicesAvailability` property on the AppConnect object.
2. Calls the `-appConnect:secureServicesAvailabilityChangedTo:` method, which provides the new value in its parameter.

Your app then changes its use of secure services and takes appropriate logic paths depending on the passed `ACSecureServicesAvailability` value. For example, if secure services become unavailable, an app should:

- stop doing any secure file I/O. The app might also notify the user of limited functionality. Limited functionality is necessary because files created using secure file I/O cannot be read or updated with regular file I/O.
- disable or enable special user interfaces for copying to the pasteboard if the pasteboard policy is `ACPASTEBOARDPOLICY_SECURECOPY`. Copying to the pasteboard fails if secure services are unavailable and the pasteboard policy is `ACPASTEBOARDPOLICY_SECURECOPY`.

The -appConnect:secureFileIOPolicyChangedTo: callback method

You optionally implement this method, which is in the AppConnectDelegate protocol:

```
-(void) appConnect:(AppConnect *)appConnect secureFileIOPolicyChangedTo:
           (ACSecureFileIOPolicy)newSecureFileIOPolicy;
```

Implement this method only if your app uses secure file I/O AppConnect APIs.

When the secure file I/O policy changes, the AppConnect library:

1. Updates the `secureFileIOPolicy` property on the AppConnect object.
2. Calls the `-appConnect:secureFileIOPolicyChangedTo:` method, which provides the new value in its parameter.

Your app then changes its use of secure file I/O and takes appropriate logic paths depending on the passed `ACSecureFileIOPolicy` value.

The -secureFileIOPolicyApplied:message: acknowledgment method

After your app processes the information provided in the callback method, it must call this acknowledgment method on the AppConnect singleton:

```
-(void)secureFileIOPolicyApplied:(ACPolicyState)policyState message:(NSString *)message;
```

Your app passes the following parameters to this method:

- the `ACPolicyState` value that represents the success or failure of handling the secure file I/O policy update. Pass the value `ACPOLICY_APPLIED` if the app successfully handled the update. Otherwise, pass the value `ACPOLICY_ERROR`. Pass the value `ACPOLICY_UNSUPPORTED` if your app does not support secure file I/O. If you do not implement the `-secureFileIOPolicyApplied:message` method, the AppConnect singleton behaves as if you passed it `ACPOLICY_UNSUPPORTED`.
- an `NSString` explaining the `ACPolicyState` value.

Typically, you use this string to report the reason the app failed to apply the secure file I/O policy update. The string is reported in the MobileIron server log files.

Version property

```
+(NSString *)version;
```

The read-only `version` class property on the AppConnect class contains an `NSString` value. The value reflects the version of the AppConnect library that the app is working with.

A best practice is to report the AppConnect library version number on your app's About page. This information is useful to support organizations if a device user has any issues with the app.

Use the following to get the version value:

```
[AppConnect version]
```

Getting upload status for tunneled HTTP/S requests

The AppConnect library and the MobileIron client app are responsible for tunneling network connections using AppTunnel with HTTP/S tunneling.

The AppConnect for iOS SDK includes APIs that provide upload status for HTTP/S requests. ***Use these APIs only if your app uses both of the following***:

- the AppTunnel with HTTP/S tunneling feature
- the NSURLConnectionDataDelegate method
  `-connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:`
  or the NSURLSessionTaskDelegateMethod
  `-URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:`
  These methods provide upload status for HTTP/S requests.

AppConnect library behavior when using AppTunnel

Apps that access enterprise servers using NSURLConnection or NSURLSession can use AppTunnel with HTTP/S tunneling, as described in AppTunnel. The AppConnect library determines which URLs to tunnel based on the MobileIron server configuration, and creates the secure tunnel for HTTP/S requests to and responses from a server behind an organization's firewall.

One aspect of this AppTunnel handling is that the AppConnect library intercepts the following NSURLConnectionDataDelegate and NSURLSession methods:

```
-connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:
```

```
-URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:
```

This interception means that if your app has implemented the NSURLConnectionDataDelegate or NSURLSessionTaskDelegate protocol, your implementation of these methods is never called. If the app depends on this method, for example, to show the progress of an HTTP/S upload, that functionality will not work properly.

Therefore, the AppConnect for iOS SDK provides APIs to support showing the progress of an HTTP/S upload when using AppTunnel with HTTP/S tunneling.

Upload status API overview

The AppConnect for iOS API provides a mechanism for the app to receive the upload status when an HTTP/S request is tunneled using AppTunnel with HTTP/S tunneling. The mechanism uses:

- The AppConnectNetworkingDelegate protocol that you implement to receive HTTP/S upload progress data
- A category method called `-setNetworkingDelegate:` in the Networking category on the AppConnect interface. The app uses `-setNetworkingDelegate:` to tell the AppConnect object about the object of the class that implements the AppConnectNetworkingDelegate protocol.

The protocol and the category are defined in AppConnect+Networking.h.

The AppConnectNetworkingDelegate protocol

Implement the AppConnectNetworkingDelegate protocol to receive HTTP/S upload progress data about tunneled requests. This protocol contains one method that provides an estimate of the progress of the upload.

```
-(void) uploadProgressForConnectionWithURLRequest:(NSURLRequest *)request
                        bytesWritten:(NSInteger)bytesWritten
                        totalBytesWritten:(NSInteger)totalBytesWritten
                       totalBytesExpectedToWrite:(NSInteger)totalBytesExpectedToWrite;
```

The AppConnect library calls this method after it intercepts the following NSURLConnectionDataDelegate or NSURLSessionTaskDelegate methods:

```
-connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:
```

```
-URLSession:task:didSendBodyData:totalBytesSent:totalBytesExpectedToSend:
```

The AppConnectNetworkingDelegate protocol method provides the following information in its parameters:

- the number of bytes written in the latest write
- the total number of bytes written for the connection with this request
- the number of bytes the connection expects to write

The -setNetworkingDelegate: method

The Networking category of the AppConnect interface provides the method `-setNetworkingDelegate:`. If your app requires HTTP/S upload progress data on tunneled HTTP/S requests, call this method before sending the HTTP/S request.

For example:

```
[[AppConnect sharedInstance] setNetworkingDelegate:myNetworkingDelegate];
```

where myNetworkingDelegate is an instance of a class that implements the AppConnectNetworkingDelegate protocol.

Caching tunneled URL responses

Apps that access enterprise servers using NSURLSession can use AppTunnel with HTTP/S tunneling, as described in AppTunnel. By default, for a tunneled URL request:

- The data for the URL is reloaded from the originating source. Any existing locally cached response is ignored.
- The data in the response is not stored in the local cache.

The reason that AppTunnel with HTTP/S tunneling does not use locally cached responses is to avoid caching sensitive enterprise server data on the device.

However, some apps have requirements to use locally cached responses. Some examples are:

- The app requires a response even when the device has no network connectivity.
- The app requires a customized response.

If your app requires locally cached responses for URL requests that use AppTunnel with HTTP/S tunneling, use the following method, which is on the AppConnect singleton object:

```
-(void)allowLocalCachingForTunneledRequests:(BOOL)flag;
```

The value of `flag` has the following impact:

- `true`
  Allows caching for requests and responses that use AppTunnel with HTTP/S tunneling. However, whether caching actually occurs depends on the cache policy for the NSURLRequest.
- `false`
  Clears all cached responses, including responses for URL requests not using AppTunnel with HTTP/S tunneling.

IMPORTANT: Do not cache sensitive data.

AppConnectUIApplication class

Using your own UIApplication subclass

If your app uses its own subclass of UIApplication, derive your subclass from AppConnectUIApplication instead of UIApplication. Information on subclassing AppConnectUIApplication, provided in AppConnectUIApplication.h, is in Using your own UIApplication subclass.

originalDelegate property (deprecated)

NOTE: Most apps have no reason to use this property.

The AppConnectUIApplication class also provides one property:

```
@property(nonatomic, readonly) id<UIApplicationDelegate> originalDelegate;
```

The AppConnect library depends on knowing about application life cycle events, such as when the application becomes active. Requiring the app to pass every life cycle event to the AppConnect library would be too much of a burden on the app. Therefore, the AppConnect library installs a UIApplicationDelegate proxy. This proxy sits between the UIApplication and your application's UIApplicationDelegate.

Your application does not do anything to support the proxy. Use your UIApplicationDelegate as you normally would:

- The AppConnect library does not filter or modify any messages sent by iOS to the UIApplicationDelegate.
- You can still add custom methods to your UIApplicationDelegate. Call the custom method as you normally would, such as in the following statement:
  `[[UIApplication sharedApplication] delegate] customMethod];`
  The proxy passes the method invocation to your UIApplicationDelegate.
- You can set a new UIApplicationDelegate as you normally would:
  `[[UIApplication sharedApplication] setDelegate:myOtherAppDelegate];`

However, until AppConnect 4.0 for iOS, a side effect of the proxy was that the following expression did not return your UIApplicationDelegate object:

```
[[UIApplication sharedApplication] delegate]
```

Therefore, the `originalDelegate` property was available to return your UIApplicationDelegate object. Using this property is no longer necessary because the above expression now **does** return your UIApplicationDelegate object.

Note The Following:

Another side effect of the proxy was that the following expression did not return your UIApplicationDelegate's class:

```
[[[UIApplication sharedApplication] delegate] class]
```

Instead, it returned the proxy class. Therefore, using `isKindofClass:` was necessary. For example, the following returned `YES`:

```
[[UIApplication sharedApplication] isKindOfClass:[MyAppDelegate class]]
```

Encryption keys for custom cryptography

Overview of encryption keys for custom cryptography

The AppConnect library provides methods to obtain keys useful for cryptographic operations. It can provide two types of keys:

- App keys, which are keys that are specific to your app on the device
- Shared keys, which are keys that are shared among all AppConnect apps on the device

If your app requires cryptography but the AppConnect secure file I/O APIs are not sufficient, it can use an app key with custom cryptographic routines. For example, consider an app that currently relies on the iOS keychain for secure storage. The keychain is not secure if the device lacks a device passcode. Refactoring the app to use the secure file I/O APIs is possibly prohibitively difficult. Therefore, instead of refactoring, you can add code to encrypt data being stored in the keychain, and you can use an app key as the encryption key.

If your app shares encrypted data with another AppConnect app, you can use a shared key as the encryption key.

Use one of the following methods, which are on the AppConnect singleton object:

```
-(nullable ACSensitiveData *)derivedAppKeyWithIdentifier:(NSString *)identifier
                error:(NSError **)error;

-(nullable ACSensitiveData *)derivedSharedKeyWithIdentifier:(NSString *)identifier
                error:(NSError **)error;
```

If successful, these methods return an ACSensitiveData object containing a 32-byte key. These methods are successful only when secure services are available.

Related topics

Securing sensitive data such as encryption keys

The -derivedAppKeyWithIdentifier:error: method

The `-derivedAppKeyWithIdentifier:error:` method derives an encryption key. This "app key" is unique to this instance of your app. Calling this method with the same identifier in an instance of your app on a different device derives a different app key.

The app key is based on the following:

- the app
- the identifier passed as an argument
  The identifier is any string. Use a different identifier for each encryption purpose. For example, if your app uses AES, SHA-1, and HMAC routines, use a different identifier for each. Reusing an identifier for different encryption purposes weakens the key, making it more vulnerable to brute force attacks.
- a secure services seed
  Each device has one secure services seed, which is generated by the MobileIron client app. The seed is lost if the device user deletes the MobileIron client app, or the MobileIron server retires the device. The MobileIron client app does not back up the seed, so a backup and restore of the device will also cause the seed to be lost.

The -derivedSharedKeyWithIdentifier:error: method

By using the shared key provided by this method, more than one AppConnect app on the same device can share encrypted data. For example, one AppConnect app can encrypt data using a derived shared key created with a particular identifier. Another AppConnect app can then use the same identifier to get the shared key, and decrypt the data with the shared key.

The `-derivedSharedKeyWithIdentifier:error:` method derives an encryption key based on the following:

- the identifier passed as an argument
- a secure services seed
  Each device has one secure services seed, which is generated by the MobileIron client app. The seed is lost if the device user deletes the MobileIron client app, or the MobileIron server retires the device. The MobileIron client app does not back up the seed, so a backup and restore of the device will also cause the seed to be lost.

Error returns for derived key methods

When unsuccessful, the `-derivedAppKeyWithIdentifier:error:` and `-derivedSharedKeyWithIdentifier:error:` methods return an NSError object as shown in the following table:

TABLE 18. NSERROR OBJECTS RETURNED BY DERIVED KEY METHODS

NSError domain	NSError code	Description
ACErrorDomain	`ACErrorNoKeys`	Secure services are not available.
ACErrorDomain	`ACErrorInvalidArg`	The `identifier` argument is `nil` or has zero-length.

The NSError domain and code values are defined in ACError.h.

Deprecated custom cryptography methods

The following methods are deprecated.

- The -derivedAppKey:withIndex: method (deprecated)
- The -derivedSharedKey:withIndex: method (deprecated)

MobileIron recommends you use instead the `-derivedAppKeyWithIdentifier:error:` and `derivedSharedKeyWithIdentifier:error:` methods.

The -derivedAppKey:withIndex: method (deprecated)

The `-derivedAppKey:withIndex:` method derives an encryption key. This app key is unique to this instance of your app. Calling this method with the same index in an instance of your app on a different device derives a different app key.

The app key is based on the following:

- the app
- the index passed as an argument
  The index is any string. Use a different index for each encryption purpose. For example, if your app uses AES, SHA-1, and HMAC routines, use a different index for each. Reusing an index for different encryption purposes weakens the key, making it more vulnerable to brute force attacks.
- a secure services seed
  Each device has one secure services seed, which is generated by the MobileIron client app. The seed is lost if the device user deletes the MobileIron client app, or the MobileIron server retires the device. The MobileIron client app does not back up the seed, so a backup and restore of the device will also cause the seed to be lost.

The -derivedSharedKey:withIndex: method (deprecated)

The `-derivedSharedKey:withIndex:` method derives an encryption key based on the following:

- the index passed as an argument
- a secure services seed
  Each device has one secure services seed, which is generated by the MobileIron client app. The seed is lost if the device user deletes the MobileIron client app, or the MobileIron server retires the device. The MobileIron client app does not back up the seed, so a backup and restore of the device will also cause the seed to be lost.

By using a shared key, more than one AppConnect app on the same device can share encrypted data. For example, one AppConnect app can use a derived shared key with a particular index to encrypt data. Another AppConnect app can then get the same derived shared key by using the same index to decrypt the data.

Securing sensitive data such as encryption keys

For heightened security of especially sensitive data, such as encryption keys and passwords, you can use the classes ACSensitiveData or ACSensitiveMutableData. These classes use the Apple hardware known as Secure

Enclave. By using these classes, you reduce the sensitive data's attack surface, because the sensitive data is stored in the Secure Enclave rather than in plain-text in memory. Without these classes, sensitive data such as keys are stored in memory, and therefore can be captured in a memory dump.

To benefit from these classes, the device must:

- have Apple's Secure Enclave hardware.

Devices that have biometric security have Secure Enclave hardware.

- be running iOS 11 through the most recently released version as supported by MobileIron
- be running Mobile@Work 9.8 for iOS through the most recently released version as supported by MobileIron

MobileIron Go does not support this feature.

Securing sensitive data involves the following:

- Coding your app to secure sensitive data
- Configuring the MobileIron server to secure sensitive data for your app
- Debugging ACSensitiveData usage

Coding your app to secure sensitive data

The interfaces to use are:

```
@interface ACSensitiveData : NSData

@interface ACSensitiveMutubleData : ACSensitiveData

@interface ACSensitiveDataContainer : NSObject
```

These interfaces are defined in ACSensitiveData.h.

To secure your data, such as encryption keys, create an ACSensitiveData or ACSensitiveMutableData object and populate it with your sensitive data.

For data that you want to keep for a long time period, create an ACSensitiveDataContainer to hold the ACSensitiveData or ACSensitiveMutableData object.

Objective-C example

```
ACSensitiveData *key = [ACSensitiveData dataWithBytes:keyData.bytes length:keyData.length];

ACSensitiveDataContainer *containerizedKey =
             [ACSensitiveDataContainer containerWithData:key];
```

Swift example

```
let key = ACSensitiveData(bytes: keyData.bytes, length: UInt(keyData.length))

let containerizedKey = ACSensitiveDataContainer(data: key)
```

NOTE: Do not use the ACSensitiveData methods `-copyWithZone: or -mutableCopyWithZone:`.

Configuring the MobileIron server to secure sensitive data for your app

The MobileIron server administrator configures whether the ACSensitiveData and ACSensitiveMutableData objects are secured with Apple's Secure Enclave. The administrator configures this choice per AppConnect app. Therefore, in the documentation you provide the MobileIron server administrators, specify that your app uses the Secure Enclave if it is available.

The MobileIron server administrator uses the key named MI_AC_CONTAINER_TYPE in the app's app configuration. The AppConnect library consumes this key. It is not passed to your app in Its configuration key-value pairs.

The possible values for MI_AC_CONTAINER_TYPE are:

Value	Description
ENCLAVE	ACSensitiveData and ACSensitiveMutableData objects are stored in the Secure Enclave, if available on the device.
LOCAL	ACSensitiveData and ACSensitiveMutableData objects are not stored in the Secure Enclave.

Debugging ACSensitiveData usage

Because it is a hardware feature, you cannot test Secure Enclave usage in the iOS simulator. However, when running in debug mode on a device or in the iOS simulator, you can use the following environment variables on your app to check if your ACSensitiveData objects are being held in memory for too long. The value of MI_AC_CONTAINER_TYPE has no impact on using these environment variables.

- AC_SENSITIVE_DATA_MAX_LIFETIME
  Set its value to a number of seconds. An exception is raised if an ACSensitiveData or ACSensitiveMutableData object is not deallocated before the specified number of seconds since its allocation. The call stack points to where the object was allocated.
- AC_SENSITIVE_DATA_MAX_RUN_LOOP_ITERATIONS
  Set its value to a positive integer. An exception is raised if an ACSensitiveData or ACSensitiveMutableData object is not deallocated before the run loop in which is was allocated completes the specified number of iterations. The call stack points to where the object was allocated.

iOS active state change notifications due to AppConnect control switches

Control switches from an AppConnect app to the MobileIron client app and then back to the app in certain situations. You can receive notifications when the app is about to move from or to the iOS active state due to these AppConnect control switches.

Use these notifications to preserve your app's state before it resigns from the iOS active state, and restore your app's state when it moves back to the iOS active state. For example, if your app is in full screen mode, preserve that fact so that the app can return to full screen mode.

Implement the following optional callback methods in the AppConnectDelegate protocol, defined in AppConnect.h:

```
-(void) applicationWillResignActiveForAppConnect:(AppConnect *)appConnect;
-(void) applicationDidBecomeActiveFromAppConnect:(AppConnect *)appConnect;
```

Situations that trigger the state change notifications

The following situations trigger the iOS active state change notifications:

- The app checkin interval expires while an AppConnect app is running. The MobileIron client app gets AppConnect policy updates for all the AppConnect apps, and then control switches back to the app that was running.
- The auto-lock time expires while an AppConnect is running.

Note that the following conditions also cause control to switch to the MobileIron client app, but do not trigger the state change notifications:

- the first time an app is launched
- the first time an app is relaunched after iOS terminated it
- after the device is powered on and the device user first launches an AppConnect app.
- after the device user logs out of secure apps in the MobileIron client app, and then relaunches an AppConnect app.

Furthermore, if control switches to the MobileIron client app, but, due to user actions, does not directly switch back to the app, `-applicationDidBecomeActiveFromAppConnect:` is not triggered. For example, `-applicationDidBecomeActiveFromAppConnect:` is not triggered if control switches from the app to the MobileIron client app because the auto-lock time expires, but the user presses the Home button instead of entering the AppConnect passcode.

Secure file I/O API details

The AppConnect for iOS SDK provides the following types of secure file I/O APIs:

- POSIX-style secure file APIs
- ACFileHandle class for AppConnect secure file I/O
- Objective-C categories for AppConnect secure file I/O

These APIs:

- Encrypt all file contents when writing, and decrypt the contents when reading.
- Allow an app to share encrypted files with other AppConnect apps. See Secure file I/O API details.
- Fail if secure services are not available. See Secure services API details.
- Some of the ACFileHandle secure methods and some of the category methods take a pointer to an NSError object as a parameter. See NSError objects that secure Objective-C methods return.

Note The Following:

- **Do not use other file I/O methods on a file** if you use AppConnect secure file I/O methods on the file.
  When you use secure file I/O APIs on a file, the first step is always to create the file using a secure file I/O API. After that, use only secure file I/O APIs on the file.
  Using both AppConnect secure file I/O methods and other file I/O methods can sometimes irreparably corrupt the files. You can use both POSIX-style AppConnect secure file I/O methods and the AppConnect secure file Objective-C subclass and category methods.
- Do not use AppConnect secure file I/O methods on a file if it contains no secure information.
  Apps that write secure data sometimes also write data that does not need to be secured. For example, user settings and preferences typically do not need to be secured. Use regular file I/O methods to save this information.
- Do not use AppConnect secure file I/O methods to read files bundled with you app, such as strings files, images, and plists.

POSIX-style secure file APIs

To secure the contents of your app's files, your Objective-C or Swift app can use C-language, POSIX-style, AppConnect secure file APIs declared in ACSecureFile.h. These APIs:

- Work only on regular files.
  They do not work on directories, pipes, named pipes, character specials, block specials, or symbolic links.
- Encrypt all file contents when writing, and decrypt the contents when reading.
- Have the same parameters, return types, and functionality as their corresponding POSIX APIs, but with the added encryption and decryption capabilities.
  For information on the corresponding POSIX APIs, see, for example, the sections "Standard I/O Streams" and "System Interfaces" at:
  http://pubs.opengroup.org/onlinepubs/009696699/functions/contents.html
- Fail if secure services are not available.
- Provide additional error information besides setting `errno`.

The following table shows each secure file I/O API and its corresponding POSIX API:

TABLE 19. SECURE FILE I/O API AND CORRESPONDING POSIX API

Secure File I/O API	Corresponding POSIX API
ACSecureFileClose()	close()
ACSecureFileLseek()	lseek()
ACSecureFileOpen()	open()

TABLE 19. SECURE FILE I/O API AND CORRESPONDING POSIX API (CONT.)

Secure File I/O API	Corresponding POSIX API
ACSecureFilePread()	pread()
ACSecureFilePwrite()	pwrite()
ACSecureFileRead()	read()
ACSecureFileReadv()	readv()
ACSecureFileRename()	rename()
ACSecureFileWrite()	write()
ACSecureFileWritev()	writev()
ACSecureFstat()	fstat()
ACSecureFtruncate()	ftruncate()
ACSecureLstat()	lstat()
ACSecureTruncate()	truncate()

Additional error returns using ACSecureFileLastError()

The secure file I/O APIs add a layer on top of the POSIX APIs to provide encryption. This layer allows the secure file I/O APIs to provide more detailed error information than available in `errno`. This additional error information is available through the method `ACSecureFileLastError()`, defined in ACSecureFile.h:

```
int ACSecureFileLastError(int fd);
```

You can call this method when:

- a POSIX-style secure file I/O API has failed.
- the failed method operated on a valid and open file descriptor.

The `ACSecureFileLastError()` method returns one of the following enumeration values, defined in ACError.h:

TABLE 20. ACSECUREFILELASTERROR() RETURN VALUES

Return value	Description
`ACE_NO_ERROR`	No error occurred.
`ACE_NO_KEYS_ERROR`	AppConnect encryption keys are not available. This error occurs when secure services are not available. See The secureServicesAvailability and secureFileIOPolicy properties .
`ACE_FILE_TOO_BIG_ERROR`	The operation would result in exceeding the maximum file size, which

TABLE 20. ACSECUREFILELASTERROR() RETURN VALUES (CONT.)

Return value	Description
	is 6,961,618,944 bytes.
ACE_NEGATIVE_FILE_LEN_ERROR	The operation would result in a negative file size.
ACE_LOW_MEMORY_ERROR	A memory alloc failed while trying to perform the operation.
ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR	An encryption operation failed, due to either a corrupt encryption key or other corrupt data. Some situations that can cause this error are: • The device user has uninstalled and reinstalled the MobileIron client app, and re-registered it with the MobileIron server. • Mixing secure and regular file routines on a file.
ACE_INVALID_ARG	One of the arguments had an invalid value.
ACE_REGULAR_FILE_ONLY_ERROR	An NSURL parameter is not a file URL. The operation is allowed only on regular files.
ACE_INTERNAL_ERROR	An error occurred in the encryption layer of the function. The file is possibly no longer accessible.

The following table shows which secure file I/O APIs set these additional error values:

TABLE 21. ADDITIONAL RETURN VALUES SET BY SECURE FILE I/O APIS

Secure File I/O API	Sets these additional return values
ACSecureFileClose()	None
ACSecureFileLseek()	• ACE_NO_KEYS_ERROR • ACE_FILE_TOO_BIG_ERROR • ACE_NEGATIVE_FILE_LEN_ERROR • ACE_INVALID_ARG
ACSecureFileOpen()	None
ACSecureFilePread()	• ACE_NO_KEYS_ERROR • ACE_READ_ON_WRITEONLY_ERROR • ACE_INTERNAL_ERROR • ACE_LOW_MEMORY_ERROR • ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR
ACSecureFilePwrite()	• ACE_NO_KEYS_ERROR • ACE_WRITE_ON_READONLY_ERROR • ACE_FILE_TOO_BIG_ERROR • ACE_LOW_MEMORY_ERROR • ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR

TABLE 21. ADDITIONAL RETURN VALUES SET BY SECURE FILE I/O APIS (CONT.)

Secure File I/O API	Sets these additional return values
	• ACE_INTERNAL_ERROR
ACSecureFileRead()	• ACE_NO_KEYS_ERROR • ACE_READ_ON_WRITEONLY_ERROR • ACE_INTERNAL_ERROR • ACE_LOW_MEMORY_ERROR • ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR
ACSecureFileReadv()	• ACE_NO_KEYS_ERROR • ACE_READ_ON_WRITEONLY_ERROR • ACE_LOW_MEMORY_ERROR • ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR • ACE_INTERNAL_ERROR
ACSecureFileRename()	None
ACSecureFileWrite()	• ACE_NO_KEYS_ERROR • ACE_WRITE_ON_READONLY_ERROR • ACE_FILE_TOO_BIG_ERROR • ACE_LOW_MEMORY_ERROR • ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR • ACE_INTERNAL_ERROR
ACSecureFileWritev()	• ACE_NO_KEYS_ERROR • ACE_WRITE_ON_READONLY_ERROR • ACE_FILE_TOO_BIG_ERROR • ACE_LOW_MEMORY_ERROR • ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR • ACE_INTERNAL_ERROR
ACSecureFstat()	• ACE_NO_KEYS_ERROR • ACE_INTERNAL_ERROR
ACSecureFtruncate()	• ACE_NO_KEYS_ERROR • ACE_TRUNC_ON_READONLY_ERROR • ACE_FILE_TOO_BIG_ERROR • ACE_NEGATIVE_FILE_LEN_ERROR • ACE_LOW_MEMORY_ERROR • ACE_INTERNAL_ERROR
ACSecureLstat()	None
ACSecureTruncate()	None

ACFileHandle class for AppConnect secure file I/O

The AppConnect for iOS SDK provides one Objective-C subclass for secure file I/O:

```
@interface ACFileHandle:NSFileHandle
```

ACFileHandle is declared in ACFileHandle.h.

To secure the contents of your app's files, your app can use ACFileHandle instead of NSFileHandle. Note that ACFileHandle:

- Works only on regular files.
  It does not work on directories, sockets, pipes, or devices as NSFileHandle does.
- Overrides most of the NSFileHandle methods, encrypting all file contents when writing, and decrypting the contents when reading.
- Adds methods to support a special error indicating that the encryption key is not available. These methods encrypt all file contents when writing, and decrypt when reading.
  Each of these added methods correspond to an overridden method. The difference is that the added method takes a pointer to an NSError object as a parameter.

NOTE: Always use the added method that has an NSError parameter rather than the corresponding overridden method. The NSError parameter allows you to code the error handling necessary when the encryption key is not available.

- Does not support asynchronous file I/O.
  ACFileHandle does not override the methods of NSFileHandle related to asynchronous I/O. Calling one of the NSFileHandle asynchronous I/O methods on a ACFileHandle object throws an exception.
- Cannot be used if secure services are not available.

Overridden and added NSFileHandle methods

ACFileHandle overrides many methods of NSFileHandle to provide secure file I/O. It also adds methods corresponding to overridden methods to support an NSError parameter. The NSError parameter allows you to code the error handling necessary when the encryption key is not available.

The following table lists the overridden and added methods. Use the methods just as you would use the corresponding NSFileHandle methods, with the differences given in the table.

NOTE: If an overridden method has a corresponding added method that includes an NSError parameter, always use the added method.

TABLE 22. NSFILEHANDLE OVERRIDDEN AND ADDED METHODS

Overridden and added methods	Usage differences with NSFileHandle method
`+ (id) fileHandleForReadingAtPath:(NSString *)path;`	The `path` parameter must be a regular file. NOTE: Do not use. Use instead the corresponding added method that includes an NSError parameter.
`+ (id) fileHandleForReadingAtPath:(NSString *)path error:(NSError *__autoreleasing *) error;`	The `path` parameter must be a regular file. Adds an NSError parameter.
`+ (id) fileHandleForReadingFromURL:(NSURL *)url error:(NSError *__autoreleasing *) error;`	The `url` parameter must be a file URL, and point to a regular file.
`+ (id) fileHandleForUpdatingAtPath:(NSString *)path;`	The `path` parameter must be a regular file. NOTE: Do not use. Use instead the corresponding added method that includes an NSError parameter.
`+ (id) fileHandleForUpdatingAtPath:(NSString *)path error:(NSError *__autoreleasing *) error;`	The `path` parameter must be a regular file. Adds an NSError parameter.
`+ (id) fileHandleForUpdatingURL:(NSURL *)url error:(NSError *__autoreleasing *) error;`	The `url` parameter must be a file URL, and point to a regular file.
`+ (id) fileHandleForWritingAtPath:(NSString *)path;`	The `path` parameter must be a regular file. NOTE: Do not use. Use instead the corresponding added method that includes an NSError parameter.
`+ (id) fileHandleForWritingAtPath:(NSString *)path error:(NSError *__autoreleasing *) error;`	The `path` parameter must be a regular file. Adds an NSError parameter.
`+ (id) fileHandleForWritingToURL:(NSURL *)url error:(NSError *__autoreleasing *)error;`	The `url` parameter must be a file URL, and point to a regular file.
`- (NSData *) availableData;`	No usage differences. NOTE: Do not use. Use instead the corresponding added method that includes an NSError

TABLE 22. NSFILEHANDLE OVERRIDDEN AND ADDED METHODS (CONT.)

Overridden and added methods	Usage differences with NSFileHandle method
	parameter.
`- (NSData *) availableDataWithError: (NSError *__autoreleasing *)error;`	Adds an NSError parameter.
`- (NSData *) readDataToEndOfFile;`	No usage differences. NOTE: Do not use. Use instead the corresponding added method that includes an NSError parameter.
`- (NSData *) readDataToEndOfFileWithError: (NSError *__autoreleasing *)error;`	Adds an NSError parameter.
`- (NSData *) readDataOfLength:(NSUInteger) length;`	No usage differences. NOTE: Do not use. Use instead the corresponding added method that includes an NSError parameter.
`- (NSData *) readDataOfLength:(NSUInteger) length (NSError *__autoreleasing *)error;`	Adds an NSError parameter.
`- (void) writeData:(NSData *) data;`	No usage differences. NOTE: Do not use. Use instead the corresponding added method that includes an NSError parameter.
`- (void) writeData:(NSData *) data (NSError *__autoreleasing *)error;`	Adds an NSError parameter.
`- (unsigned long long) offsetInFile;`	No usage differences.
`- (unsigned long long) seekToEndOfFile;`	No usage differences.
`- (void) seekToFileOffset: (unsigned long long)offset;`	No usage differences.

TABLE 22. NSFILEHANDLE OVERRIDDEN AND ADDED METHODS (CONT.)

Overridden and added methods	Usage differences with NSFileHandle method
`- (void) truncateFileAtOffset: (unsigned long long)offset;`	No usage differences.
`- (void) synchronizeFile;`	No usage differences.
`- (void) closeFile;`	No usage differences.

ACFileHandle example

The following example makes a secure copy of an unsecured file. Specifically, the example:

1. Creates a secure file.
2. Writes the contents of the unsecured file /etc/group into it.
3. Reads the contents of the secure file.

NOTE: For brevity, the example does not include error handling.

```
- (void) ACFileHandleExample
{
	NSString *secureFileName = @"/tmp/secureGroup";
	NSError *error;

	// Read the contents of /etc/group
	NSData *etcGroupData = [NSData dataWithContentsOfFile:@"/etc/group"];

	// Use the default file manager to create the secure file.
	[[NSFileManager defaultManager] createFileAtPath:secureFileName
	                                        contents:nil attributes:nil];

	// Get the file handle for writing to the secure file just created.
	ACFileHandle *acFileHandle =
	                [ACFileHandle fileHandleForWritingAtPath:secureFileName error:&error];

	// Write the contents of /etc/group to the secure file.
	[acFileHandle writeData:etcGroupData error:&error];

	// Close the secure file.
	[acFileHandle closeFile];

	// Open the secure file for reading.
	acFileHandle = [ACFileHandle fileHandleForReadingAtPath:secureFileName error:&error];

	//read entire contents of the secure file.
	NSData *duplicate = [acFileHandle readDataToEndOfFileWithError:&error];

	// Close the secure file
	[acFileHandle closeFile];
```

```
        // Note: The contents of NSData objects 'duplicate' and 'etcGroupData'
        // are identical.
}
```

Objective-C categories for AppConnect secure file I/O

The AppConnect for iOS SDK provides the following categories, in which each method corresponds to a method in the original class, but provides a secure version of the functionality.

- NSFileManager category
- NSData (ACSecureFile) category
- NSData (ACSharedSecureFile) and ACFileHandle (ACSharedSecureFile) categories
- NSKeyedArchiver category
- NSKeyedUnarchiver category
- NSDictionary category
- NSMutableDictionary category
- NSArray category
- NSMutableArray category

Note The Following:

- These methods cannot be used if secure services are not available.
- These methods provide a special error indicating that the encryption key is not available.
  Methods that take a pointer to an NSError object as a parameter provide this error indication. See NSError objects that secure Objective-C methods return.
- The header files are in the AppConnect.framework in <category name>.h.

NSFileManager category

Each method in the NSFileManager category corresponds to a method in the NSFileManager class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSFileManager in developer.apple.com.

NOTE: The category methods return an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSFileManager.

TABLE 23. NSFILEMANAGER CATEGORY METHODS

Method in category	Corresponding method in NSFileManager
- (BOOL)createSecureFileAtPath: (NSString *)path contents:(NSData *)contents attributes:(NSDictionary *)attributes error:(NSError *__autoreleasing *)error;	- (BOOL)createFileAtPath: (NSString *)path contents:(NSData *)contents attributes:(NSDictionary *)attributes;
- (BOOL)moveSecureFileAtPath: (NSString *)srcPath toPath:(NSString *)dstPath error:(NSError *__autoreleasing *)error;	- (BOOL)moveItemAtPath: (NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)moveSecureFileAtURL: (NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError *__autoreleasing *)error;	- (BOOL)moveItemAtURL: (NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
- (BOOL)copySecureFileAtPath: (NSString *)srcPath toPath:(NSString *)dstPath error:(NSError *__autoreleasing *)error;	- (BOOL)copyItemAtPath: (NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)copySecureFileAtURL: (NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError *__autoreleasing *)error;	- (BOOL)copyItemAtURL: (NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;

TABLE 23. NSFILEMANAGER CATEGORY METHODS (CONT.)

Method in category	Corresponding method in NSFileManager
- (NSData *)secureContentsAtPath: (NSString *)path error:(NSError *__autoreleasing *)error;	- (NSData *)contentsAtPath: (NSString *)path;
- (BOOL)secureContentsEqualAtPath: (NSString *)path1 andPath:(NSString *)path2 error:(NSError *__autoreleasing *)error;	- (BOOL)contentsEqualAtPath: (NSString *)path1 andPath:(NSString *)path2;
- (NSDictionary *) attributesOfSecureFileAtPath: (NSString *)path error:(NSError *__autoreleasing *)error;	- (NSDictionary *) attributesOfItemAtPath: (NSString *)path error:(NSError **)error;

Example:

The following example shows how to move a secure file to a new location. Specifically, the example:

1. Creates a secure file.
2. Writes the contents of the unsecured file /etc/group into the secure file.
3. Moves the secure file to a new location using the NSFileManager+ACSecureData category methods.

NOTE: For brevity, the example does not include error handling.

```
- (void)NSFileManagerCategoryExample
{
	NSError *error;

	// Read the contents of /etc/group.
	NSData *etcGroupData = [NSData dataWithContentsOfFile:@"/etc/group"];

	// Create a secure file with the contents of /etc/group.
	NSString *secureFileName = @"/tmp/secureFile";
	[[NSFileManager defaultManager] createSecureFileAtPath:secureFileName
                                                contents:etcGroupData attributes:nil];

	// Move the newly created secure file to a new location.
	// First, create the source and destination file URLs.
	NSString *anotherSecureFileName = @"/tmp/anotherSecureFile";
	NSURL *sourceURL = [NSURL fileURLWithPath:secureFileName];
```

```
        NSURL *destinationURL = [NSURL fileURLWithPath:anotherSecureFileName];

        // Move the secure file.
        [[NSFileManager defaultManager] moveSecureFileAtURL:sourceURL
                                                      toURL:destinationURL error:&error];

        // Note: The following line incorrectly moves a secure file.
        // Mixing regular and secure file I/O on the same file can result
        // in corrupted data.
        // DO NOT USE.
        // [[NSFileManager defaultManager] moveItemAtPath:sourceURL toPath:destinationURL
                                                                  error:&error];
}
```

NSData (ACSecureFile) category

Use this category if you to encrypt the data that your app stores. If you want to share the encrypted data with another AppConnect app, see NSData (ACSecureFile) category.

Each method in the NSData (ACSecureFile) category corresponds to a method in the NSData class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSData in developer.apple.com.

Note The Following:

- The `url` parameter in the category methods must be a file URL, and point to a regular file.
- The category methods that return an NSError object set the properties on the object as described in NSError objects that secure Objective-C methods return.
- MobileIron recommends that you only use the category methods that return an NSError object. However, to be consistent with the NSData class, the category includes secure versions of NSData methods that do not return an NSError object.

The following table shows each added method and its corresponding method in NSData.

TABLE 24. NSDATA (ACSECUREFILE) CATEGORY METHODS

Method in category	Corresponding method in NSData
+ (id)dataWithContentsOfSecureFile: (NSString *)path;	+ (id)dataWithContentsOfFile: (NSString *)path;
+ (id)dataWithContentsOfSecureFile: (NSString *)path options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;	+ (id)dataWithContentsOfFile: (NSString *)path options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;
+ (id)dataWithContentsOfSecureURL: (NSURL *)url;	+ (id)dataWithContentsOfURL: (NSURL *)url;
+ (id)dataWithContentsOfSecureURL: (NSURL *)url options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;	+ (id)dataWithContentsOfURL: (NSURL *)url options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;
- (id)initWithContentsOfSecureFile: (NSString *)path;	- (id)initWithContentsOfFile: (NSString *)path;
- (id)initWithContentsOfSecureFile: (NSString *)path options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;	- (id)initWithContentsOfFile: (NSString *)path options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;
- (id)initWithContentsOfSecureURL: (NSURL *)url;	- (id)initWithContentsOfURL: (NSURL *)url;
- (id)initWithContentsOfSecureURL: (NSURL *)url options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;	- (id)initWithContentsOfURL: (NSURL *)url options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;
- (BOOL)writeToSecureFile: (NSString *)path atomically:(BOOL)flag;	- (BOOL)writeToFile: (NSString *)path atomically:(BOOL)flag;

TABLE 24. NSDATA (ACSECUREFILE) CATEGORY METHODS (CONT.)

Method in category	Corresponding method in NSData
- (BOOL)writeToSecureFile: (NSString *)path options: (NSDataWritingOptions)mask error:(NSError **)errorPtr;	- (BOOL)writeToFile: (NSString *)path options: (NSDataWritingOptions)mask error:(NSError **)errorPtr;
- (BOOL)writeToSecureURL: (NSURL *)aURL atomically:(BOOL)atomically;	- (BOOL)writeToURL: (NSURL *)aURL atomically:(BOOL)atomically;
- (BOOL)writeToSecureURL: (NSURL *)aURL options: (NSDataWritingOptions)mask error:(NSError **)errorPtr;	- (BOOL)writeToURL: (NSURL *)aURL options: (NSDataWritingOptions)mask error:(NSError **)errorPtr;

Example:

The following example shows how to use NSData category methods to:

1. Create a secure file and write data to it.
2. Read the contents of the secure file.

NOTE: For brevity, the example does not include error handling.

```
- (void)NSDataCategoryExample
{
       NSError *error;

       // Read the contents of /etc/group.
       NSData *etcGroupData = [NSData dataWithContentsOfFile:@"/etc/group"];

       // Write the contents of /etc/group to a secure file.
       NSString *secureFileName = @"/tmp/group.sec";
       [etcGroupData writeToSecureFile:secureFileName options:0 error:&error];

       // Read the contents of the secure file.
       NSData *secureFileData =
              [NSData dataWithContentsOfSecureFile:secureFileName options:0 error:&error];

       // Note: The contents of NSData objects 'secureFileData' and 'etcGroupData'
       // are identical.
}
```

NSData (ACSharedSecureFile) and ACFileHandle (ACSharedSecureFile) categories

Use these categories if you want to encrypt the data that your app stores **and** you want the app to share the data with another AppConnect app. An encryption group ID determines which apps can share encrypted data. Each method in these categories corresponds to a method in NSData or NSFileHandle, and includes an encryption group ID parameter. The methods use the encryption group ID when encrypting and decrypting data. Therefore, any app using the same encryption group ID can share the encrypted data.

Note The Following:

- If you do not want to share the data with another AppConnect app, use NSData (ACSharedSecureFile) and ACFileHandle (ACSharedSecureFile) categories and NSData (ACSharedSecureFile) and ACFileHandle (ACSharedSecureFile) categories.
- If you want to share data from a Document View Controller extension to a host app, see Sharing secure files from an extension.

Your app receives the encryption group ID in its app-specific configuration key-value pairs. Therefore, to use these categories, do the following:

1. Define the encryption group ID key name that your app expects to receive in its app-specific configuration. For example: `com.sample.groupID`

The number of characters in the key name is not limited.

2. Include information about the key in your documentation for MobileIron server administrators. The information includes:
   - The name of the key
   - The other AppConnect apps that are sharing the encrypted data
     Each of these other AppConnect apps also do these steps.
3. Handle receiving app-specific configuration as described in App-specific configuration API details .
4. Use the value of the encryption group ID key received in the app-specific configuration in the methods of these categories.

Note The Following:

- The `url` parameter in these categories' methods must be a file URL, and point to a regular file.
- The categories' methods that return an NSError object set the properties on the object as described in NSError objects that secure Objective-C methods return.
- MobileIron recommends that you only use the methods that return an NSError object. However, to be consistent with the NSData and NSFileHandle classes, these categories include secure versions of NSData and ACFileHandle methods that do not return an NSError object.

The following table shows each added method for NSData(ACSharedSecureFile) and its corresponding method in NSData.

TABLE 25. NSDATA(ACSHAREDSECUREFILE) CATEGORY METHODS

Method in category	Corresponding method in NSData
`+ (id)dataWithContentsOfSecureFile: (NSString *)path encryptionGroupId:(NSString *)groupId;`	`+ (id)dataWithContentsOfFile: (NSString *)path;`
`+ (id)dataWithContentsOfSecureFile: (NSString *)path encryptionGroupId:(NSString *)groupId options:(NSDataReadingOptions)mask error:(NSError **)errorPtr;`	`+ (id)dataWithContentsOfFile: (NSString *)path options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;`
`+ (id)dataWithContentsOfSecureURL: (NSURL *)url encryptionGroupId:(NSString *)groupId;`	`+ (id)dataWithContentsOfURL: (NSURL *)url;`
`+ (id)dataWithContentsOfSecureURL: (NSURL *)url encryptionGroupId:(NSString *)groupId options:(NSDataReadingOptions)mask error:(NSError **)errorPtr;`	`+ (id)dataWithContentsOfURL: (NSURL *)url options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;`
`- (id)initWithContentsOfSecureFile: (NSString *)path encryptionGroupId:(NSString *)groupId;`	`- (id)initWithContentsOfFile: (NSString *)path;`
`- (id)initWithContentsOfSecureFile: (NSString *)path encryptionGroupId:(NSString *)groupId options:(NSDataReadingOptions)mask error:(NSError **)errorPtr;`	`- (id)initWithContentsOfFile: (NSString *)path options: (NSDataReadingOptions)mask error:(NSError **)errorPtr;`
`- (id)initWithContentsOfSecureURL: (NSURL *)url encryptionGroupId:(NSString *)groupId;`	`- (id)initWithContentsOfURL: (NSURL *)url;`
`- (id)initWithContentsOfSecureURL: (NSURL *)url`	`- (id)initWithContentsOfURL: (NSURL *)url`

TABLE 25. NSDATA(ACSHAREDSECUREFILE) CATEGORY METHODS (CONT.)

Method in category	Corresponding method in NSData
encryptionGroupId:(NSString *)groupId options:(NSDataReadingOptions)mask error:(NSError **)errorPtr;	options:(NSDataReadingOptions)mask error:(NSError **)errorPtr;
- (BOOL)writeToSecureFile: (NSString *)path encryptionGroupId:(NSString *)groupId atomically:(BOOL)flag;	- (BOOL)writeToFile: (NSString *)path atomically:(BOOL)flag;
- (BOOL)writeToSecureFile: (NSString *)path encryptionGroupId:(NSString *)groupId options:(NSDataWritingOptions)mask error:(NSError **)errorPtr;	- (BOOL)writeToFile: (NSString *)path options:(NSDataWritingOptions)mask error:(NSError **)errorPtr;
- (BOOL)writeToSecureURL: (NSURL *)aURL encryptionGroupId:(NSString *)groupId atomically:(BOOL)atomically;	- (BOOL)writeToURL: (NSURL *)aURL atomically:(BOOL)atomically;
- (BOOL)writeToSecureURL: (NSURL *)aURL encryptionGroupId:(NSString *)groupId options:(NSDataWritingOptions)mask error:(NSError **)errorPtr;	- (BOOL)writeToURL: (NSURL *)aURL options:(NSDataWritingOptions)mask error:(NSError **)errorPtr;

Example using NSData(ACSharedSecureFile) category methods:

The following example shows how to use NSData(ACSharedSecureFile) category methods to:

1. Create a shared secure file and write data to it.
2. Read the contents of the secure file.

NOTE: For brevity, the example does not include error handling.

```
- (void)NSDataSharedCategoryExample
{
       NSError *error;

       // This example assumes the app has already:
```

```
        // 1. Retrieved the encryption group Id value from the config property on
        //    the AppConnect object.
        // 2. Stored the value in an NSString * property named groupId of the current object.

        // Read the contents of /etc/group.
        NSData *etcGroupData = [NSData dataWithContentsOfFile:@"/etc/group"];

        // Write the contents of /etc/group to a secure file to be shared with
        // another AppConnect app.
        NSString *secureFileName = @"/tmp/group.sec";
        [etcGroupData writeToSecureFile:secureFileName
                encryptionGroupId:self.groupId
                options:0 error:&error];

        // Read the contents of the secure file.
                NSData *secureFileData =
                       [NSData dataWithContentsOfSecureFile:secureFileName
                               encryptionGroupId:self.groupId
                               options:0 error:&error];

        // Note: The contents of NSData objects 'secureFileData' and 'etcGroupData'
        // are identical.
}
```

Example using ACFileHandle(ACSharedSecureFile) category methods:

The following example shows how to use ACFileHandle(ACSharedSecureFile) category methods to:

1. Create a shared secure file and write data to it.
2. Read the encrypted contents of the secure file, decrypt the contents, and write it to a unsecured file.

NOTE: For brevity, the example does not include error handling.

```
- (void)ACFileHandleSharedCategoryExample
{
        NSError *error;

        // This example assumes the app has already:
        // - Retrieved the encryption group Id value from the config property on
        //   the AppConnect object.
        // - Stored the value in an NSString * property named groupId of the current object.
        // - Stored URLs in NSString * properties destinationPathURL and decryptedURL
        //   of the current object.

        // Read the contents of /etc/group.
        NSError *err;
        NSFileHandle *sourceFileHandle =
            [NSFileHandle fileHandleForReadingAtPath:@"/etc/group" error:&err];

        // Get a file handle to a file to share with another AppConnect app.
        ACFileHandle *destFileHandle =
             [ACFileHandle fileHandleForWritingToURL:self.destinationPathURL
                                                withEncryptionGroupId:self.groupID
                                                error:&err];
```

```
        //Read chunks and write them using the secure file handle.
        NSData *data = nil;
        while ((data = [sourceFileHandle readDataOfLength:1024]) && (data.length > 0)) {
            [destFileHandle writeData:data error:&Serr];
            NSLog(@"Wrote bytes (%@)", err.description);
        }
        [destFileHandle synchronizeFile];

        // Read the contents of the secure file.
        ACFileHandle *sharedEncryptedFileHandle =
        [ACFileHandle fileHandleForReadingFromURL:self.destinationPathURL
                              withEncryptionGroupId:self.groupID
                              error:&err];

        // Create an empty file.
        [[NSFileManager defaultManager] createFileAtPath:self.decryptedURL.path
                                        contents:nil
                                        attributes:nil];

        // Read the encrypted file, decrypt the data, and write it to an unencrypted file.
        NSFileHandle *writeToFileHandle =
             [NSFileHandle fileHandleForWritingAtPath:@"/etc/group-copy"];
        NSData *decryptedData = nil;
        while ((decryptedData = [sharedEncryptedFileHandle readDataOfLength:1024]) &&
                                                              (decryptedData.length > 0)) {
             [writeToFileHandle writeData:decryptedData];
        }
        [writeToFileHandle synchronizeFile];
        // Note: The contents of @"/etc/group" and @"/etc/group-copy" are identical.
}
```

NSKeyedArchiver category

Each method in the NSKeyedArchiver category corresponds to a method in the NSKeyedArchiver class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSKeyedArchiver in developer.apple.com.

NOTE: The category methods return an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSKeyedArchiver.

TABLE 26. NSKEYEDARCHIVER CATEGORY METHODS

Method in category	Corresponding method in NSKeyedArchiver

`+ (BOOL)archiveRootObject:`
`(id)rootObject`
`toSecureFile:(NSString *)path`
`error:(`
`NSError *__autoreleasing *)error;`
	`+ (BOOL)archiveRootObject:`
	`(id)rootObject`
	`toFile:(NSString *)path;`

`+ (BOOL)archiveRootObject:`
`(id)rootObject`
`toSecureFile:(NSString *)path`
`atomically:(BOOL)atomically`
`error:(`
`NSError *__autoreleasing *)error;`
	`+ (BOOL)archiveRootObject:`
	`(id)rootObject`
	`toFile:(NSString *)path`
	`atomically:(BOOL)atomically;`

Example:

The following example shows how to use NSKeyedArchiver and NSKeyedUnarchiver category methods to:

1. Create a secure archive file and write data to it from a mutable dictionary.
2. Read the contents of the secure archive file into another mutable dictionary.

NOTE: For brevity, the example does not include error handling.

```
- (void)NSKeyedArchiverCategoryExample
{
       NSError *error;

       // Create and populate a mutable dictionary.
       NSMutableDictionary *dict = [NSMutableDictionary dictionary];

       NSString *key1 = @"baseball";
       NSString *value1 = @"white";
       [dict setValue:value1 forKey:key1];

       NSString *key2 = @"basketball";
       NSString *value2 = @"orange";
       [dict setValue:value2 forKey:key2];

       // Archive the dictionary to a secure file.
       NSString *archiveName = @"/tmp/secureArchive";
                   [NSKeyedArchiver archiveRootObject:dict toSecureFile:archiveName error:&error];

       // Unarchive the secure file contents into another dictionary.
       NSMutableDictionary *dictCopy = (NSMutableDictionary*)[NSKeyedUnarchiver
                                       unarchiveObjectWithSecureFile:archiveName error:&error];
```

```
        // Note: The contents of NSMutableDictionary objects 'dict' and 'dictCopy'
        // are identical.

}
```

NSKeyedUnarchiver category

Each method in the NSKeyedUnarchiver category corresponds to a method in the NSKeyedUnarchiver class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSKeyedUnarchiver in developer.apple.com.

NOTE: The category method returns an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSKeyedUnarchiver.

TABLE 27. NSKEYEDUNARCHIVER CATEGORY METHODS

Method in category	Corresponding method in NSKeyedUnarchiver
`+ (id)unarchiveObjectWithSecureFile: (NSString *)path error:( NSError *__autoreleasing *)error;`	`+ (id)unarchiveObjectWithFile: (NSString *)path;`

For a code example, see NSKeyedUnarchiver category.

NSDictionary category

Each method in the NSDictionary category corresponds to a method in the NSDictionary class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSDictionary in developer.apple.com.

Note The Following:

- The `url` parameter in the category methods must be a file URL, and point to a regular file.
- The category methods return an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSDictionary.

TABLE 28. NSDICTIONARY CATEGORY METHODS

Method in category	Corresponding method in NSDictionary

`dictionaryWithContentsOfSecureFile:`
`+ (id)  (NSString *)path`
`  error:(`
`  NSError *__autoreleasing *)error;`
	`+ (id)dictionaryWithContentsOfFile:`
	`  (NSString *)path;`

`+ (id)`
`dictionaryWithContentsOfSecureURL:`
`  (NSURL *)aURL`
`  error:(`
`  NSError *__autoreleasing *)error;`
	`+ (id)dictionaryWithContentsOfURL:`
	`  (NSURL *)aURL;`

`- (id)initWithContentsOfSecureFile:`
`  (NSString *)path`
`  error:(`
`  NSError *__autoreleasing *)error;`
	`- (id)initWithContentsOfFile:`
	`  (NSString *)path;`

`- (id)initWithContentsOfSecureURL:`
`  (NSURL *)aURL`
`  error:(`
`  NSError *__autoreleasing *)error;`
	`- (id)initWithContentsOfURL:`
	`  (NSURL *)aURL;`

`- (BOOL)writeToSecureFile:`
`  (NSString *)path`
`  atomically:(BOOL)flag`
`  error:(`
`  NSError *__autoreleasing *)error;`
	`- (BOOL)writeToFile:`
	`  (NSString *)path`
	`  atomically:(BOOL)flag;`

`- (BOOL)writeToSecureURL:`
`  (NSURL *)aURL`
`  atomically:(BOOL)flag`
`  error:(`
`  NSError *__autoreleasing *)error;`
	`- (BOOL)writeToURL:`
	`  (NSURL *)aURL`
	`  atomically:(BOOL)flag;`

Example:

The following example shows how to use NSDictionary and NSMutableDictionary category methods to:

1. Create a secure file and write data to it from a NSMutableDictionary object.
2. Read the contents of the secure file into an NSDictionary object.

NOTE: For brevity, the example does not include error handling.

```
- (void)NSDictionaryCategoryExample
{
       NSError *error;

       // Create and populate a dictionary.
       NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:@"baseball",
                                    @"white", @"basketball", @"orange", nil];

       // Write the dictionary to a secure file.
       NSString *secureFileName = @"/tmp/secureDictionary";
       [dict writeToSecureFile:secureFileName atomically:TRUE error:&error];

       // Create a dictionary with the contents of the secure file.
       NSDictionary *dictCopy = [[NSDictionary alloc]
                                    initWithContentsOfSecureFile:secureFileName error:&error];

       // Note: The contents of objects 'dict' and 'dictCopy' are identical.
}
```

NSMutableDictionary category

Each method in the NSMutableDictionary category corresponds to a method in the NSMutableDictionary class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSMutableDictionary in developer.apple.com.

Note The Following:

- The `url` parameter in the category methods must be a file URL, and point to a regular file.
- The category methods return an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSMutableDictionary.

TABLE 29. NSMUTABLEDICTIONARY CATEGORY METHODS

Method in category	Corresponding method in NSMutableDictionary

`+ (id)`
 	
`dictionaryWithContentsOfSecureFile:`
`(NSString *)path`
`error:(`
`NSError *__autoreleasing *)error;`
	`+ (id)dictionaryWithContentsOfFile:`
	`(NSString *)path;`

`+ (id)`
 	
`dictionaryWithContentsOfSecureURL:`
`(NSURL *)aURL`
`error:(`
`NSError *__autoreleasing *)error;`
	`+ (id)dictionaryWithContentsOfURL:`
	`(NSURL *)aURL;`

`- (id)initWithContentsOfSecureFile:`
`(NSString *)path`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (id)initWithContentsOfFile:`
	`(NSString *)path;`

`- (id)initWithContentsOfSecureURL:`
`(NSURL *)aURL`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (id)initWithContentsOfURL:`
	`(NSURL *)aURL;`

`- (BOOL)writeToSecureFile:`
`(NSString *)path`
`atomically:(BOOL)flag`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (BOOL)writeToFile:`
	`(NSString *)path`
	`atomically:(BOOL)flag;`

`- (BOOL)writeToSecureURL:`
`(NSURL *)aURL`
`atomically:(BOOL)flag`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (BOOL)writeToURL:`
	`(NSURL *)aURL`
	`atomically:(BOOL)flag;`

For a code example, see NSMutableDictionary category.

NSArray category

Each method in the NSArray category corresponds to a method in the NSArray class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSArray in developer.apple.com.

Note The Following:

- The `url` parameter in the category methods must be a file URL, and point to a regular file.
- The category methods return an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSArray.

TABLE 30. NSARRAY CATEGORY METHODS

Method in category	Corresponding method in NSArray
`+ (id) arrayWithContentsOfSecureFile: (NSString *)path error:( NSError *__autoreleasing *)error;`	`+ (id)arrayWithContentsOfFile: (NSString *)path;`
`+ (id) arrayWithContentsOfSecureURL: (NSURL *)aURL error:( NSError *__autoreleasing *)error;`	`+ (id)arrayWithContentsOfURL: (NSURL *)aURL;`
`- (id)initWithContentsOfSecureFile: (NSString *)path error:( NSError *__autoreleasing *)error;`	`- (id)initWithContentsOfFile: (NSString *)path;`

TABLE 30. NSARRAY CATEGORY METHODS (CONT.)

Method in category	Corresponding method in NSArray

`- (id)initWithContentsOfSecureURL:`
`(NSURL *)aURL`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (id)initWithContentsOfURL:`
	`(NSURL *)aURL;`

`- (BOOL)writeToSecureFile:`
`(NSString *)path`
`atomically:(BOOL)flag`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (BOOL)writeToFile:`
	`(NSString *)path`
	`atomically:(BOOL)flag;`

`- (BOOL)writeToSecureURL:`
`(NSURL *)aURL`
`atomically:(BOOL)flag`
`error:(`
`NSError *__autoreleasing *)error;`
	`- (BOOL)writeToURL:`
	`(NSURL *)aURL`
	`atomically:(BOOL)flag;`

Example:

The following example shows how to use NSArray and NSMutableArray category methods to:

1. Create a secure file and write data to it from a NSMutableArray object.
2. Read the contents of the secure file into an NSArray object.

NOTE: For brevity, the example does not include error handling.

```
- (void)NSArrayCategoryExample
{
       NSError *error;

       // Create an array and populate it.
        NSArray *array = [NSArray arrayWithObjects:@"one fish", @"two fish", @"red fish",
                                                   @"blue fish", nil];

       // Write the array to a secure file.
       NSString *secureArrayFileName = @"/tmp/secureArray";
       [array writeToSecureFile:secureArrayFileName atomically:TRUE error:&error];

       // Create an array from the contents of the secure file.
       NSArray *arrayCopy = [[NSArray alloc]
                             initWithContentsOfSecureFile:secureArrayFileName error:&error];

       // The contents of the objects 'array' and 'arrayCopy' are identical.
}
```

NSMutableArray category

Each method in the NSMutableArray category corresponds to a method in the NSMutableArray class, but provides a secure version of the functionality. For more information about the functionality and usage, see NSMutableArray in developer.apple.com.

Note The Following:

- The `url` parameter in the category methods must be a file URL, and point to a regular file.
- The category methods return an NSError object. The methods set the properties on the object as described in NSError objects that secure Objective-C methods return.

The following table shows each added method and its corresponding method in NSMutableArray.

TABLE 31. NSMUTABLEARRAY CATEGORY METHODS

Method in category	Corresponding method in NSMutableArray
`+ (id) arrayWithContentsOfSecureFile: (NSString *)path error:( NSError *__autoreleasing *)error;`	`+ (id)arrayWithContentsOfFile: (NSString *)path;`
`+ (id) arrayWithContentsOfSecureURL: (NSURL *)aURL error:( NSError *__autoreleasing *)error;`	`+ (id)arrayWithContentsOfURL: (NSURL *)aURL;`
`- (id)initWithContentsOfSecureFile: (NSString *)path error:( NSError *__autoreleasing *)error;`	`- (id)initWithContentsOfFile: (NSString *)path;`

TABLE 31. NSMUTABLEARRAY CATEGORY METHODS (CONT.)

Method in category	Corresponding method in NSMutableArray
`- (id)initWithContentsOfSecureURL: (NSURL *)aURL error:( NSError *__autoreleasing *)error;`	`- (id)initWithContentsOfURL: (NSURL *)aURL;`
`- (BOOL)writeToSecureFile: (NSString *)path atomically:(BOOL)flag error:( NSError *__autoreleasing *)error;`	`- (BOOL)writeToFile: (NSString *)path atomically:(BOOL)flag;`
`- (BOOL)writeToSecureURL: (NSURL *)aURL atomically:(BOOL)flag error:( NSError *__autoreleasing *)error;`	`- (BOOL)writeToURL: (NSURL *)aURL atomically:(BOOL)flag;`

For a code example, see NSMutableArray category.

NSError objects that secure Objective-C methods return

Some of the ACFileHandle secure methods and some of the category methods take a pointer to an NSError object as a parameter. These methods can set the domain and code properties on the NSError object to:

- the domain NSPOSIXErrorDomain, with the `code` property set to `errno` values.
- other domains, such as NSCocoaErrorDomain. The possible values of the `code` property are the same as regular Objective-C methods.
- the domain ACErrorDomain, defined in ACError.h. The possible values of the `code` property are defined in the enumeration in ACError.h. These values are the same values returned by the `ACSecureFileLastError()` method.
  Of particular interest when working with secure file I/O APIs are the errors `ACE_NO_KEYS_ERROR` and `ACE_BAD_KEY_OR_CORRUPT_DATA_ERROR`. These errors indicate an encryption failure.
  For more information, see NSError objects that secure Objective-C methods return.

Objective-C example

The following example shows how to check the NSError object returned in a secure write method:

```
- (void)errorHandlingExample
{
    // Create data to be securely stored.
    NSData *data = [@"secret data" dataUsingEncoding:NSASCIIStringEncoding];

    // Set up a couple of data writing options.
```

```
    NSDataWritingOptions options = NSDataWritingAtomic | NSDataWritingFileProtectionComplete;

    NSString *secureFilename = @"/tmp/data.sec";
    NSError *error;

       if (! [data writeToSecureFile:secureFilename options:options error:&error]) {

               if ([[error domain] isEqualToString:ACErrorDomain] &&
                                              [error code] == ACE_NO_KEYS_ERROR) {

                  // Provide logic to handle the situation when
                  // the encryption key is not available.
               }
       }
}
```

Swift example

The following example shows how to check the NSError object returned in a secure write method:

```
func errorHandlingExample() {

       // Create data to be securely stored.
       let data = "secret data".data(using: .ascii)! as NSData

       // Set up a couple of data writing options.
       let options: NSData.WritingOptions = [.atomic, .completeFileProtection]

       let secureFilename = "/tmp/data.sec"

       do {
               try data.write(toSecureFile: secureFilename, options: options)
       }

       catch(let error as NSError) {

               if (error.domain == ACErrorDomain && error.code == ACErrorNoKeys) {

                       // Provide logic to handle the situation when
                       // the encryption key is not available.
               }
       }
}
```

Sharing secure files from an extension

An AppConnect app can provide an app extension, specifically a Document View Controller extension, to share secure files with other AppConnect apps. A file can be shared with all AppConnect apps or with only specific AppConnect apps.

NOTE: To share secure files between AppConnect apps, see Secure file I/O API details.

Sharing secure documents from an extension requires the following tasks:

- Setting up the MobileIron server for sharing files from an extension
- Setting up the provider app's Info.plist
- Coding the provider app to share secure files with its extension
- Coding the extension to share files with the host app
- Coding the host app to access the shared file

The sample app SwiftFileSharing illustrates coding these tasks in Swift.

Setting up the MobileIron server for sharing files from an extension

If you want your AppConnect app's extension to share secure files with other AppConnect apps, define values for the keys MI_AC_SHARED_GROUP_ID and MI_AC_ACCESS_CONTROL_ID. In the documentation that you provide to the MobileIron server administrator about your AppConnect, include:

- the values you define
- the AppConnect apps that you want to use your extension to access the secure files

The server administrator sets the key-value pairs in the app configuration of your app and each AppConnect app that is to share the secure files. If the server administrator does not set MI_AC_SHARED_GROUP_ID, then all AppConnect apps can access the shared secure files.

NOTE: In the MobileIron Core Admin Portal, app key-value pairs are set up in **Policy & Configs > Configurations**, in the **App-specific Configurations** section of an **AppConnect App Configuration**. In the MobileIron Cloud Admin Portal, the key-value pairs are set up in the AppConnect Custom Configuration section of the app.

Setting up the provider app's Info.plist

For a provider app to share secure files through its extension, do the following:

1. Include the following key-value pairs in the app's Info.plist:
   - MI_APP_CONNECT
     This key is the root key, and its value is a dictionary of key-value pairs
   - MI_AC_KEYCHAIN_ACCESS_GROUP
     This key provides a keychain access group that the AppConnect library uses to share secure files between the provider app and its extension. The value is the app's identifier prefix followed by a string you define.

   For example:

2. In the Xcode project, in **Capabilities > Keychain Sharing**, add the string you defined. In this example, the string to add is **com.mycompany.MyACSharedFiles**.

Coding the provider app to share secure files with its extension

The following sample code illustrates the AppConnect APIs that the provider app uses to share secure files with its extension. The sample code is followed by a table of the tasks involved.

```
// When the AppConnect isReady notification is triggered, enable extension support.
-(void)appConnectIsReady:(AppConnect *)appConnect {
       [[AppConnect sharedInstance] enableAppExtensionSupport];
}

// Insert code to use the read-only config property on the AppConnect singleton to
// get the MI_AC_SHARED_GROUP_ID key-value pair, if available.

// In this example, the key-value pair was not included, so nil
// is passed to -getCryptoKeysForACFileEncryptionWithSharedGroupID:error: for the group ID.

// When secure services are available, create an encryption key for encrypting secure files.

-(void)appConnect:(AppConnect *)appConnect
       secureServicesAvailabilityChangedTo:(ACSecureServicesAvailability)secureServicesAvailability
{

       if (secureServicesAvailability == ACSECURESERVICESAVAILABILITY_AVAILABLE) {
              NSData *secureKeyData =
                 [ACWrappedAppKey getCryptoKeysForACFileEncryptionWithSharedGroupID:nil error:nil];

              // The secureKeyData object contains the encryption key.
              // Store the secureKeyData object in a shared keychain that the extension
              // can access.
       }
}
```

TABLE 32. CODING THE PROVIDER APP TO SHARE SECURE FILES WITH ITS EXTENSION

Task	AppConnect APIs
1. Enable extension support.	Call the `-enableAppExtensionSupport:` method on the AppConnect singleton object. **Header** • AppConnectInterface.h
2. Get the value of the MI_AC_SHARED_GROUP_ID key-value pair.	Use the read-only `config` property on the AppConnect singleton to get the MI_AC_SHARED_GROUP_ID key-value pair, if available. **Related topics and header files** • App-specific configuration API details • AppConnectInterface.h
3. Make sure secure services are available.	Check if the `secureServicesAvailability` property on the AppConnect singleton has the value `ACSECURESERVICESAVAILABILITY_AVAILABLE`. Continue only if secure services are available **Related topics and header files** • Secure services API details • AppConnectInterface.h
4. Create an	**Method**

TABLE 32. CODING THE PROVIDER APP TO SHARE SECURE FILES WITH ITS EXTENSION (CONT.)

Task	AppConnect APIs
encryption key for encrypting shared files.	`+(NSData *) getCryptoKeysForACFileEncryptionWithSharedGroupID:(NSString *)groupID error:(NSError *_autoreleasing *)error;` **Parameters** • `groupID` Pass the value of the MI_AC_SHARED_GROUP_ID key-value pair. If this key-value pair is not available, pass `nil`. Passing `nil` means that all AppConnect apps can decrypt the shared file. • `error` If the method fails to create an encryption key, `error` is set to the appropriate NSError object. **Return value** • NSData object containing key used for shared file encryption Header file • ACWrappedAppKey.h
5. Store the returned encryption key in a shared keychain item used by the provider app and its extension.	

Coding the extension to share files with the host app

The following sample code illustrates what the Document View Controller extension does to share secure files with a host app. The sample code is followed by a table of the tasks involved.

```
// Add the following ExtensionManager class to your extension code. Your extension will
// create a singleton instance of the class, which takes care of all the
// AppConnect-related operations.

@class ExtensionManager;

@protocol ExtensionManagerProtocol
-(void)extensionManager:(ExtensionManager *)extensionManager
                        appConnectAccessControlStateDeterminedAs:(ACExtensionAccessState)state;
@end

@interface ExtensionManager: NSObject <AppConnectExtensionInterfaceProtocol>
```

```
@property (weak) AppConnectExtensionInterface *acInterface;
@property (weak) id<ExtensionManagerProtocol> delegate;
@end

@implementation ExtensionManager

+(instancetype)sharedInstance {
       static ExtensionManager *sharedInstance = nil;
       static dispatch_once_t onceToken;
       dispatch_once(&onceToken, ^{
           sharedInstance = [[ExtensionManager alloc] init];
           sharedInstance.acInterface = [AppConnectExtensionInterface appConnectExtensionInstance];
           sharedInstance.acInterface.delegate = sharedInstance;
       });
       return sharedInstance;
}

-(void)requestAccessControlState {
       // Initiate a process that determines whether the host app is allowed to
       // access the extension.
        [self.acInterface determineAccessControlState];
}

-(void)appConnectAccessControlStateDeterminedAs:(ACExtensionAccessState)state {
    [self.delegate extensionManager:self appConnectAccessControlStateDeterminedAs:state];
}

@end


//
//
// In your UIDocumentPickerExtensionViewController implementation, include the following
code:
//
//

-(void)prepareForPresentationInMode:(UIDocumentPickerMode)mode {

        // Insert code to present a view controller appropriate for the picker mode.
        // Then...

    switch (mode) {
       case UIDocumentPickerModeOpen:
       case UIDocumentPickerModeImport:

               ExtensionManager *extensionManager = [ExtensionManager sharedInstance];
               [extensionManager setDelegate:self];
               [extensionManager requestAccessControlState];

               // Start a spinner while waiting to find out if the host app is allowed
               // to access the extension.
```

```
                [_spinner startAnimating];
    }
}

//
//
// Your UIDocumentPickerExtensionViewController class implements
// the ExtensionManagerProtocol.
//

-(void)extensionManager:(ExtensionManager *)extensionManager
                          appConnectAccessControlStateDeterminedAs:(ACExtensionAccessState)state {
        currentState = state;
        [_spinner stopAnimating];

        switch (currentState) {

                case ACExtensionAccessStateNoRequest:
                        // A non-AppConnect App has launched the extension.
                        // Do not share the file. Take necessary steps, such as notifying the user.
                        break;

                case ACExtensionAccessStateNotEnabled:
                        // Either the administrator did not configure MI_AC_ACCESS_CONTROL_ID for
                        // the provider app, or the provider app has not setup access control by
                        // calling -enableAppExtensionSupport:.
                        // Do not share the file. Take necessary steps, such as notifying the user.
                        break;

                case ACExtensionAccessStateBlocked:
                        // The host app does not have access to this extension. It does not have
                        // the same MI_AC_ACCESS_CONTROL_ID as the provider app.
                        // Do not share the file. Take necessary steps, such as notifying the user.
                        break;

                case ACExtensionAccessStateNotBlocked:
                        // Share the wrapped file. An AppConnect app has launched the extension.
                        break;
        }
}
```

TABLE 33. CODING THE DOCUMENT VIEW CONTROLLER EXTENSION TO SHARE FILES WITH THE HOST APP

Task	AppConnect APIs and sample code

1. Define an ExtensionManager class that is derived from NSObject and implements the AppConnectExtensionInterfaceProtocol.	The sample code provides a full implementation of an ExtensionManager class that you can use.
	It includes the implementation of:
	• the AppConnectExtensionInterface method:
	`-(BOOL) determineAccessControlState;`
	• the AppConnectExtensionInterfaceProtocol callback method:
	`-(void) appCon-nectAccessControlStateDeterminedAs: (ACExtensionAccessState)state;`
	Header file
	• AppConnectExtensionInterface.h in the AppConnectExtension.framework

2. Your UIDocumentPickerExtensionViewController class implements the ExtensionManagerProtocol.	The sample code explains the handling of each `ACExtensionAccessState` value in the ExtensionManagerProtocol callback method. The app continues to file sharing processing only if the value is `ACExtensionAccessStateNotBlocked.`
3. Your UIDocumentPickerExtensionViewController object does the following: - Creates a singleton instance of the ExtensionManager class. - Sets the ExtensionManager's delegate so that you can receive the callback. - Initiates the request to determine if the host app is allowed to use the extension.	The sample code shows this sequence in `-prepareForPresentationInMode:.`

TABLE 33. CODING THE DOCUMENT VIEW CONTROLLER EXTENSION TO SHARE FILES WITH THE HOST APP (CONT.)

Task	AppConnect APIs and sample code
4. When the ExtensionManagerProtocol callback method is called with `state` set to `ACExtensionAccessStateNotBlocked`, read the encryption key stored as an NSData object in the shared keychain item.	
5. Wrap the selected file using the encryption key.	**Method** `+(BOOL)wrapFileAtPath:(NSString *)path toPath:(NSString *)toPath withCryptoBlock:(NSData *)cryptoBlock actualFileName:(NSString *fileName) error:(NSError *_autoreleasing *)error;` **Parameters** • `path` Pass the file URL of the selected file. • `toPath` Pass the file URL of where the resulting wrapped file should be stored. • `cryptoBlock` Pass the NSData object containing the encryption key. • `actualFileName` Optional. File name for the wrapped file, if it should be different than the original file name. • `error` If the method fails, `error` is set to the appropriate NSError object. **Return value** • `YES` if successful. Otherwise `NO`. **Header file** • ACWrappedFile.h in the AppConnectExtension.framework

Coding the host app to access the shared file

The following sample code illustrates what the host app does to access the secure file shared by the extension. The sample code is followed by a table of the tasks and header files involved.

```
// Insert code to use the read-only config property on the AppConnect singleton to
// get the MI_AC_SHARED_GROUP_ID key-value pair, if available.
// In this example, the key-value pair was not included. Therefore, nil
// is passed for the group ID parameter to -readWrappedFileAtPath:sharedGroupID:error:,
```

```
// which gets the file handle of the shared, wrapped file.

-(NSURL *)getDecryptedFileURL:(NSURL *)url {

       ACWrappedFileReadHandle *readHandle = [ACUnwrappedFile readWrappedFileAtPath:url.path
                                                                    sharedGroupID:nil error:&error];
       if (readHandle) {

              NSFileHandle *writeToFileHandle =
                   [NSFileHandle fileHandleForWritingAtPath:decURL.path];

              // Decrypt the file by reading it with the ACWrappedFileReadHandle object.
              // This snippet then writes it to an unencrypted file.

              NSData *decryptedData = nil;
              while ((decryptedData =
                      [readHandle readDataOfLength:1024]) && (decryptedData.length > 0)) {

                        [writeToFileHandle writeData:decryptedData];
              }

              [writeToFileHandle synchronizeFile];
              [writeToFileHandle closeFile];

              // You can remove the wrapped file after decrypting it.
              [[NSFileManager defaultManager] removeItemAtURL:url error:nil];
              return decURL;
       }
       else if (error && error.code == ACWrappedFileReadErrorUnknownWrapperFormat) {

              // The file is not wrapped. It is not from an AppConnect app's extension.
              // It can be used directly.
              return url;
       }
}
```

TABLE 34. CODING THE HOST APP TO ACCESS THE SHARED FILE

Task	AppConnect APIs
1. Get the value of the MI_AC_SHARED_GROUP_ID key-value pair.	Use the read-only `config` property on the AppConnect singleton to get the MI_AC_SHARED_GROUP_ID key-value pair, if available. **Related topics and header files** • App-specific configuration API details • AppConnectInterface.h
2. Get the file handle of the shared, wrapped file.	**Method** `+(ACWrappedFileReadHandle *) readWrappedFileAtPath:(NSString *)path sharedGroupID:(NSString *)groupID error:(NSError *__autoreleasing *)error;` **Parameters** • `path` Pass the file URL of the file returned from the extension. • `sharedGroupID` Pass the value of the MI_AC_SHARED_GROUP_ID key-value pair. If this key-value pair is not available, pass `nil`. • `error` If the method fails, `error` is set to the appropriate NSError object. **Return value** • If successful, returns the file handle of the shared, wrapped file as a ACWrappedFileReadHandle object. Otherwise, returns `nil`. **Header files** • ACUnwrappedFile.h in the AppConnect.framework • ACWrappedFileReadHandle.h in the AppConnect.framework
3. Using the file handle, read and decrypt the file's contents.	**Methods** Use the methods in ACFIleHandle.h to read and decrypt the file's contents.

AppTunnel diagnostic API details

The AppTunnel diagnostic API provides troubleshooting information for an app's use of AppTunnel with HTTP/S tunneling. Typically, you add a user interface, such as a menu item, to invoke a diagnostic run for tunneling to a specified URL. Your app then displays or logs the results of the diagnostic run. The MobileIron server administrator uses the results to troubleshoot AppTunnel configuration for the app.

NOTE: An AppTunnel diagnostic API is also available for Xamarin projects. See "How to include the Xamarin C# binding in your Xamarin project" in Developing AppConnect Apps with Xamarin .

The AppTunnel diagnostic API provides the following AppTunnel information:

- Whether the device received any AppTunnel rules
- Whether the URL requested matched an AppTunnel rule
- Whether a valid pinned Standalone Sentry server certificate is available for tunneling. This is the certificate that devices use to know that the Sentry used for AppTunnel is a trusted server.
- Whether a valid client identity is available. This client identity is used to authenticate the app to the Sentry.
- Whether the Sentry is reachable
- The HTTP/S status code returned from the backend server
- Whether AppTunnel is blocked
- Whether the device received data from the backend server
- Whether the backend server redirected the URL request
- Whether the backend server issued an authentication challenge

The API is defined in the Networking category of the AppConnect interface, in the header file AppConnect+Networking.h.

Running an AppTunnel diagnostic

To run an AppTunnel diagnostic, use the following method:

```
-(NSInteger)diagnoseTunnelingForURL:(NSURL *)url
              resultHandler:(void (^)(ACTunnelingDiagnosticResult *result,
                                      NSInteger runID)) resultHandler;
```

The method `-diagnoseTunnelingForURL:resultHandler:` makes successive calls to the `resulthandler` block as it progresses through the diagnostic run for the specified URL. Each call to the `resulthandler` block contains information about processing the URL for tunneling. When the diagnostic run is complete, `-diagnoseTunnelingForURL:resultHandler:` makes a final call to the `resulthandler` block, passing it `nil` for the result.

IMPORTANT: The URL request must have no side effects, such as modifying data on the server. This requirement is because if the URL request is successful, the destination server receives the request, but your app does not receive the response.

For example, the following code snippets (one for Objective-C and one for Swift):

- Passes a URL from a text field.
- Passes an in-line block to log the results of the diagnostic run.

Objective-C example

```
[[AppConnect sharedInstance] diagnoseTunnelingForURL:
               [NSURL URLWithString:self.urlField.text]
                resultHandler:^(ACTunnelingDiagnosticResult *result, NSInteger runID) {

         if (result) {
                NSLog(@"Diagnostic run %I result %@: %@", runID,
                            result.successful?@"Success":@"FAILURE",
                            result.resultDescription);
         } else {
```

```
                NSLog(@"Diagnostic run %@I ended", runID);
        }
 }];
```

Swift example

```
AppConnect.sharedInstance()?.diagnoseTunneling (
       for: URL(string: self.urlField.text)!,
            resultHandler: { (result, runID) in

               if (nil != result) {

                       print("Diagnostic run \(runID) result \(result!.isSuccessful ? "Success" :
                               "FAILURE"):  \(result!.description)")
               }
               else {
                       print("Diagnostic run \(runID) ended")
               }
       }
    )
}
```

For more information, continue to:

- -diagnoseTunnelingForURL:resultHandler: parameters
- -diagnoseTunnelingForURL:resultHandler: return value
- The result handler for diagnostic runs
- The ACTunnelingDiagnosticResult class
- The ACTunnelingDiagnosticResultCode enumeration
- AppTunnel configuration troubleshooting checklist for MobileIron Core

-diagnoseTunnelingForURL:resultHandler: parameters

The following table describes the parameters that you pass to `-diagnoseTunnelingForURL:resultHandler:`.

TABLE 35. PARAMETERS PASSED TO -DIAGNOSETUNNELINGFORURL:RESULTHANDLER

Parameter	Description
`url`	An NSURL object specifying the URL to diagnose.
`resultHandler`	A callback block that you define. It is called successive times with each result as the diagnostic run progresses.

-diagnoseTunnelingForURL:resultHandler: return value

The method `-diagnoseTunnelingForURL:resultHandler:` returns an NSInteger value. The value is the same as the value of the `runID` parameter returned to the `resulthandler` block. The `runID` is a unique number assigned by each diagnostic run. The `runID` parameter is useful for distinguishing different runs of the AppTunnel diagnostic.

For each call to `-diagnoseTunnelingForURL:resultHandler:`, you pass:

- a URL
- a `resulthandler` block

The `runID` associates the results passed back in the `resulthandler` block with the URL being diagnosed.

The result handler for diagnostic runs

In the `resulthandler` block, put the logic to handle the successive results of a diagnostic run. For example, log the values of the properties of the `result` parameter.

The `resulthandler` block has the following parameters:

TABLE 36. RESULTHANDLER BLOCK PARAMETERS

Parameter	Description
`result`	An ACTunnelingDiagnosticResult object returned with each callback to the result handler. When the diagnostic run is complete, `-diagnoseTunnelingForURL:resultHandler:` makes a final call to the `resulthandler` block, passing it `nil` for the result.
`runID`	A variable to contain the ID that associates the returned `result` with a URL. The `runID` is a unique number assigned by each diagnostic run. The `runID` parameter is useful for distinguishing different runs of the AppTunnel diagnostic. For each call to `-diagnoseTunnelingForURL:resultHandler:`, you pass: • a URL • a `resulthandler` block The `runID` associates the results passed back in the `resulthandler` block with the URL being diagnosed.

The ACTunnelingDiagnosticResult class

The ACTunnelingDiagnosticResult class represents one of the results of a diagnostic run. The result handler receives an instance of the ACTunnelingDiagnosticResult class.

The object has these properties:

TABLE 37. ACTUNNELINGDIAGNOSTICRESULT PROPERTIES

Property	Description
`resultCode`	A ACTunnelingDiagnosticResultCode value.
`successful`	`YES` if the result was successful. Otherwise, `NO`.
`timestamp`	An NSDate object containing the timestamp for when the result occurred.
`resultDescription`	A description of the result. **Important:** These descriptions are for readability only, and possibly will change in future releases. Do not depend on these strings for programmatic decisions. Use the `resultCode`.

The ACTunnelingDiagnosticResultCode enumeration

The ACTunnelingDiagnosticResultCode enumeration values are returned in each result of a diagnostic run. Not all values are returned with each run, and some values can be returned more than once in each run. For example, if a URL request is redirected more than once, `ACTDR_REDIRECT` is returned more than once.

The enumeration values fall into these categories:

TABLE 38. ACTUNNELINGDIAGNOSTICRESULTCODE ENUMERATION VALUES

Category	Description
Diagnostic run life cycle codes	Status of diagnostic run's progress.
Policy integrity codes	Information about the AppTunnel policy data for the app on the device.
Certificate challenges codes	Whether using the available certificates is successful.
Networking codes	Whether the Standalone Sentry is reachable.
Connection result codes	Information about the HTTP/S connection

The following table provides:

- the enumeration's values
- a description of each value
- the values of the `successful` and `resultDescription` properties in the ACTunnelingDiagnosticResult object.

IMPORTANT: These `resultDescription` strings are for readability only, and possibly will change in future releases. Do not depend on these strings for programmatic decisions. Use the `resultCode`.

TABLE 39. ACTUNNELINGDIAGNOSTICRESULT OBJECT DESCRIPTIONS FOR EACH RESULT CODE

Value	Details
Diagnostic run life cycle codes	
ACTDR_RUN_STARTED	The run started properly. `successful`: Always `YES`. `resultDescription`: Diagnostic run started. Requesting (URL)
ACTDR_REDIRECT	The server redirected to a new URL. `successful`: Always `YES`. `resultDescription`: Redirected by server to new URL (url)
ACTDR_COMPLETED	Indicates whether the diagnostic run completed successfully. `successful`: `YES` if completed without an error. Otherwise `NO`. `resultDescription`: Session completed normally or Session completed with error: (error)
ACTDR_ABORT_UNSUPPORTED_AUTH	The diagnostic run ended because the server issued an authentication challenge, such as basic authentication. This challenge is normally handled by the app, so the diagnostic run cannot continue. Typically, if the diagnostic run gets to this result, AppTunnel is working. `successful`: Always `YES`. `resultDescription`: Server issued an auth challenge type that the diagnostic does not support. Aborting the diagnostic and the auth challenge. Auth challenge type is (auth type)
Policy integrity codes, evaluating AppTunnel policy information for the app on the device	
ACTDR_RULE_MATCH	Indicates whether the URL matches an AppTunnel rule. If the URL redirects, another result with this code is returned for the redirection URL. `successful`: `YES` if matched. Otherwise `NO`. `resultDescription`: Request matches a tunneling rule so it will be tunneled. or Request does not match a tunneling rule so it will not be tunneled.

TABLE 39. ACTUNNELINGDIAGNOSTICRESULT OBJECT DESCRIPTIONS FOR EACH RESULT CODE (CONT.)

Value	Details
	or AppTunnel policy has no tunneling rules.
ACTDR_POLICY_SERVER_CERT	Indicates whether a valid pinned Sentry certificate is available for tunneling. This is the certificate that the AppConnect Library in the app uses to know that the Sentry used for AppTunnel is a trusted server. Note different Sentrys can provide tunneling for different URL requests. Each Sentry has its own pinned certificate. `successful`: `YES` if the certificate is valid. Otherwise `NO`. `resultDescription`: Server certificate in the AppTunnel policy is valid. or Server certificate in the AppTunnel policy is invalid. It may have expired. or No server certificate was found in the AppTunnel policy.
ACTDR_POLICY_CLIENT_IDENTITY	Indicates whether a valid client identity is available. This client identity is used to authenticate the app to the Sentry. `successful`: `YES` if the client identity is valid. Otherwise `NO`. `resultDescription`: Client identity in the AppTunnel policy appears to be valid. or Client certificate in the AppTunnel policy is invalid. It may have expired. or No client identity in the AppTunnel policy.
Certificate challenges codes, indicating whether using the certificates is successful	
ACTDR_SEND_CLIENT_CERT	Indicates whether the app successfully authenticated the app to the Sentry using the available client identity. `successful`: `YES` if authentication to the Sentry succeeded. Otherwise `NO`. `resultDescription`: Authenticated with client identity
ACTDR_EVALUATE_SENTRY_CERT	Indicates whether the Sentry passed evaluation using the pinned Sentry certificate. This is the certificate that the AppConnect Library in the app uses to know that the Sentry used for AppTunnel is a trusted

TABLE 39. ACTUNNELINGDIAGNOSTICRESULT OBJECT DESCRIPTIONS FOR EACH RESULT CODE (CONT.)

Value	Details
	server. `successful`: `YES` if the certificate is trusted. Otherwise `NO`. `resultDescription`: Server certificate passed all evaluation or Server certificate was not trusted. The trust result was (trust result)
Networking codes	
ACTDR_DNSLOOKUP_SENTRY	Indicates whether a DNS lookup for the Sentry has succeeded. `successful`: `YES` if the lookup succeeded. Otherwise `NO`. `resultDescription`: DNS resolution of the Sentry <*Sentry hostname*> succeeded or DNS resolution of the Sentry <*Sentry hostname*> failed

TABLE 39. ACTUNNELINGDIAGNOSTICRESULT OBJECT DESCRIPTIONS FOR EACH RESULT CODE (CONT.)

Value	Details
Connection result codes	
ACTDR_RESPONSE	Indicates that the server returned an HTTP status code. The value of the HTTP status code is in the `resultDescription`. `successful`: `YES` for HTTP status codes 1xx, 2xx, or 3xx. `NO` for 4xx and 5xx. `resultDescription`: Received HTTP status code (code) or AppTunnel is blocked. Note The Following: • Some HTTP status codes are handled and consumed by iOS, and therefore do not generate a callback to the result handler. • Blocking AppTunnel blocks access to web sites configured to use AppTunnel. The MobileIron administrator can block AppTunnel for a device through a manual action or an automatic action triggered by a security violation on the device.
ACTDR_RECEIVED_DATA	Indicates that data was received from the backend server in the HTTP/S response. `successful`: Always `YES`. `resultDescription`: Received (bytes) bytes of data NOTE: This result shows the bytes as they are received, not the total number of bytes.

AppTunnel configuration troubleshooting checklist for MobileIron Core

If an app is not successfully tunneling to its app server, check the following in the MobileIron Core Admin Portal:

TABLE 40. APPTUNNEL CONFIGURATION TROUBLESHOOTING CHECKLIST FOR MOBILEIRON CORE

Admin Portal location	Troubleshooting actions
Settings > Preferences	Under **Additional Products**, make sure you have enabled the appropriate features. Make sure you have selected **Enable App Tunnel for third-party and in-house apps**, if you are using AppTunnel for any app besides Docs@Work.
Policies & Configs > Policies AppConnect global policy	Check the AppConnect global policy configuration: 1. In the **AppConnect** field, make sure you have selected **Enabled**. 2. Make sure AppConnect global policy is applied to a label belonging to the device. If you are using the default AppConnect global policy, this step is not necessary. 3. If you do not create an AppConnect container policy for the app, select **Authorize** for **Apps without an AppConnect container policy**.
Settings > Sentry	Make sure the Standalone Sentry is configured with a certificate that devices use to know that the Sentry used for AppTunnel is a trusted server. To view the Sentry certificate in the Admin Portal for MobileIron Core. 1. Go to **Settings > Sentry**. 2. Find the line for the appropriate Sentry. 3. Click **View Certificate.**
Settings > Sentry	Make sure the Standalone Sentry is configured for AppTunnel for the app: 1. Make sure **Enable AppTunnel** is selected. 2. In **Device Authentication Configuration**, make sure the correct, valid Trusted Root Certificate is uploaded. 3. In **AppTunnel Configuration,** make sure you have configured the **Services**.
Policies & Configs > Configurations AppConnect container policy	Check the AppConnect container policy for the app. Make sure it is applied to a label belonging to the device. You do not need an AppConnect container policy if the AppConnect global policy selects **Authorize** for **Apps without an AppConnect container policy.**
Policies & Configs > Configurations AppConnect app configuration	Check the AppConnect app configuration for the app: 1. Make sure the **AppTunnel Rules** point to the intended Sentry and service. 2. For **Identity Certificate,** make sure you have selected the correct certificate, issued from the trusted root Certificate Authority indicated by the Trusted Root Certificate uploaded to the Sentry. 3. Make sure the certificate has not expired and that its initial validity date is in the past. 4. Make sure AppConnect app configuration is applied to a label belonging to the device.

UIScene support

With iOS 13 Apple moved `UIApplicationDelegate` events handling to `UISceneDelegate`. To function properly, AppConnect requires some of the events that are now handled by UIScene.

Therefore, if your app supports UIScene, when initializing the AppConnect library, call the AppConnect method `-sceneWillConnectToSessionWithOptions:`.

The method must be called from UISceneDelegate's `-scene:willConnectToSession:options:` method. UIScene connection options need to be passed as input parameter to the AppConnect instance method `-sceneWillConnectToSessionWithOptions:`.

The method has the parameter `options:`. The value for the parameter is the value provided to `[UISceneDelegate scene:willConnectToSession:options:]`.

See also, Initialize the AppConnect libraryHow to initialize your Xamarin app to use AppConnect C# APIs.

Best Practices Using the AppConnect for iOS SDK

The following are best practices for developing secure enterprise apps:

- Display authorization status in the home screen
- Allow the user to enter credentials manually
- Use the AppConnectDelegate protocol for notifications
- Limit the size of configuration data from the MobileIron server
- Use the UIApplication's delegate as you normally would
- Consider limitations when using the iOS simulator
- Enable the AppConnect library to blur screens when the app becomes inactive
- Do not put secure data in the app bundle
- Indicate to the user that the app is initializing
- Reject custom keyboard control
- Do not use UIWebView to upload sensitive data
- Provide documentation about your app to the MobileIron server administrator

Display authorization status in the home screen

When an app becomes unauthorized or retires, the `authState` property on the AppConnect object changes to `ACAUTHSTATE_UNAUTHORIZED` or `ACAUTHSTATE_RETIRED`. Additionally, the `authMessage` property changes to a string that explains to the device user why the app is unauthorized or retired. The string sometimes also explains what the device user can do to make the app authorized again.

The app should display the `authMessage` string. However, consider that since the app is now unauthorized or retired, the app must exit its secure functionality. Therefore, the best user experience is to display the string in a home view that never contains secure information.

The following alternatives for displaying the `authMessage` string are not recommended:

- Do not display the string using UIAlertView on top of the current view. Beneath the message, the current view can still have secure information visible.
- Do not use the `-displayMessage:` method. This method does not match the look of your app.
- Do not exit the app without displaying the string.

Allow the user to enter credentials manually

Always provide a way for a user to enter login credentials manually in your app. Provide this user interface even if you are receiving login credentials in app-specific configuration information from the AppConnect library.

As described in Configuration specific to the app, a MobileIron server administrator can set up configuration information for your app on the server. Your app receives the information using the AppConnect for iOS SDK. This information can include authentication credentials, such as username, password and certificates, for a corporate service. Because the app receives the information, the device user does not have to enter the information.

However, if the credentials change, the amount of time for the change to reach your application can vary. Some variables that impact this notification include:

- the app checkin interval that the administrator configured on the MobileIron server. This value is the maximum number of minutes until devices running AppConnect apps receive updates of their AppConnect policies and app-specific configurations.
- whether the device has network coverage.

Therefore, providing changes to devices is not a real-time process and can take up to several hours. Therefore, if the corporate service rejects the credentials, provide a way for the user to enter the credentials manually.

Use the AppConnectDelegate protocol for notifications

Use methods of the AppConnectDelegate protocol to receive notifications of changes to:

- the authorization status and associated message.
- the permission status for copying content to the iOS pasteboard, using document interaction (Open In and Open From), and print.
- app-specific configuration.

Do not use the iOS SDK's key-value observing capabilities instead of AppConnectDelegate protocol notifications.

Consider the following scenario in which the AppConnect library receives a new authorization status:

1. The `authState` property on the AppConnect object changes from `ACAUTHSTATE_AUTHORIZED` to `ACAUTHSTATE_UNAUTHORIZED`.
2. The `authMessage` property on the AppConnect object changes from "The app is authorized." to "The app is not authorized because your device OS is compromised."
3. The AppConnect library calls the `-appConnect:authStateChangedTo:withMessage` method on the AppConnectDelegate.

Now consider what can happen if you use key-value observing on the `authState` property. When `authState` changes, an application typically displays to the user the string in `authMessage`. Because the `authMessage` string

has not yet changed, the user sees the original message that “The app is authorized.” However, the app is no longer authorized.

Using only the AppConnectDelegate’s callback methods avoids such inconsistencies.

Limit the size of configuration data from the MobileIron server

Do not design your app to use large amounts of configuration data from the MobileIron server.

As described in Configuration specific to the app, a MobileIron server administrator can set up configuration information for your app on the server. Your app receives the information using the AppConnect for iOS SDK. Use this capability only for short strings and options, such as server addresses, authentication credentials, and certificates.

Do not use it for larger data items, such as documents, large blocks of HTML, or images. For large data items, use a web service to deliver the items. Use AppConnect configuration only to provide the URL for the web service.

Although no precise upper limit is defined for an item configured on the MobileIron server, a large item can impact server performance. It can also slow connectivity between the server and the MobileIron client app. A very large item can possibly cause the communication protocol between the MobileIron server and the MobileIron client app to fail entirely.

Use the UIApplication’s delegate as you normally would

The AppConnect library depends on knowing about application life cycle events, such as when the application becomes active. Requiring the app to pass every life cycle event to the AppConnect library would be too much of a burden on the app. Therefore, the AppConnect library installs a UIApplicationDelegate proxy. This proxy sits between the UIApplication and your application’s UIApplicationDelegate.

Your application does not do anything to support the proxy. Use your UIApplicationDelegate as you normally would:

- The AppConnect library does not filter or modify any messages sent by iOS to the UIApplicationDelegate.
- You can still add custom methods to your UIApplicationDelegate. Call the custom method as you normally would, such as in the following statement:
  ```
  [[UIApplication sharedApplication] delegate] customMethod];
  ```
  The proxy passes the method invocation to your UIApplicationDelegate.
- You can set a new UIApplicationDelegate as you normally would:
  ```
  [[UIApplication sharedApplication] setDelegate:myOtherAppDelegate];
  ```

Until AppConnect 4.0 for iOS, the UIApplicationDelegate proxy caused side effects. Now these side effects do not occur, The side effects of the proxy were:

- The following expression did not return your UIApplicationDelegate's class:
  `[[[UIApplication sharedApplication] delegate] class]`
  Instead, it returned the proxy class.
  Therefore, prior to AppConnect 4.0, using `isKindofClass:` was necessary. For example, the following returned `YES`:
  `[[UIApplication sharedApplication] isKindOfClass:[MyAppDelegate class]]`
- The following expression did not return your UIApplicationDelegate object:
  `[[UIApplication sharedApplication] delegate]`
  If you required access to your UIApplicationDelegate object, the AppConnectUIApplication class provided a property called `originalDelegate`. Because this property is no longer necessary, it is deprecated.
  See AppConnectUIApplication class.

Consider limitations when using the iOS simulator

To fully test an AppConnect app, debug on a tethered device using Xcode, as you would for any other app. On a device, your testing includes the MobileIron client app, which is necessary for the complete flow of data from the MobileIron server to your app.

You can do initial functionality testing in the iOS simulator in Xcode. You can link against the AppConnect library when building for the iOS simulator as you normally would.

However, when using the AppConnect library in the iOS simulator, the AppConnect library always sets the properties on the AppConnect singleton as follows:

- the `authState` property is set to `ACAUTHSTATE_AUTHORIZED`
- the `config` property has no entries
- the `pasteboard` property is set to `ACPASTEBOARDPOLICY_AUTHORIZED`
- the `openInPolicy` property is set to `ACOPENINPOLICY_AUTHORIZED`
- the `openFromPolicy` property is set to `ACOPENFROMPOLICY_AUTHORIZED`
- the `printPolicy` property is set to `ACPRINTPOLICY_AUTHORIZED`

This behavior is necessary because no simulator version of the MobileIron client app is available, and the MobileIron client app is necessary for your app to receive notifications. Without notifications, the `authState` property cannot change to `ACAUTHSTATE_AUTHORIZED`, and your app cannot execute its logic that accesses its secure data and functionality. The AppConnect library's special simulator behavior solves this problem, allowing you to use the iOS simulator to test your app's functionality. You cannot, however, use the simulator to test handling notifications from the AppConnect library.

Enable the AppConnect library to blur screens when the app becomes inactive

AppConnect 4.0 for iOS added support for blurring screens when the app becomes inactive. Use this capability of the AppConnect library, as described in Enable screen blurring. If your app provided its own screen blurring, remove that code. By using the AppConnect library's screen blurring capability, all AppConnect apps behave consistently.

Do not put secure data in the app bundle

Files that you package in your app bundle are not AppConnect-encrypted files. Also, files packaged with an app cannot be modified at runtime. Therefore, these files are not secure. Therefore, include only non-sensitive data in the app bundle.

Indicate to the user that the app is initializing

Indicate in the user interface that the app is initializing if the app requires the AppConnect singleton's instance properties to determine what to do. For example, use an activity indicator (spinner). Remove the indication after the app is notified that the AppConnect singleton is ready.

One reason this indication is important involves when to display sensitive data. Do not show any sensitive data until the AppConnect singleton is ready, because until that time, the app cannot determine whether it is authorized. Only an authorized app should show sensitive data.

Reject custom keyboard control

Custom keyboard extensions sometimes send data to servers when a device user enters data into an app. They send this data for assistance with word-prediction, for example. This behavior has potential for harmful data loss. MobileIron server administrators can control whether your app can use a custom keyboard by specifying a key-value pair (MI_AC_IOS_ALLOW_CUSTOM_KEYBOARDS) on your app's configuration. Your app can control whether custom keyboards are allowed if the server administrator has enabled the key-value pair.

To reject custom keyboards when the server administrator has enabled the key-value pair, implement the `-shouldAllowExtensionPointIdentifier:` method on your AppDelegate as follows:

```
// Reject all non-native keyboards.
- (BOOL) application:(UIApplication *) application
       shouldAllowExtensionPointIdentifier:(NSString *)extensionPointIdentifier
{

       if ([extensionPointIdentifier
              isEqualToString:UIApplicationKeyboardExtensionPointIdentifier])
```

```
        {
                return NO;
        }
        return YES;
}
```

Related topics

Custom keyboard control

Do not use UIWebView to upload sensitive data

When an app uploads data, such as a file or image, using UIWebView, the UIWebView object saves the data in a folder on the device. The folder is `Apps/<app name>/tmp`. The data is then available using, for example, iExplorer.

Therefore, apps should not use UIWebView to upload sensitive data. If you cannot change the app's use of UIWebView, be sure to delete any sensitive data from the folder after each upload attempt, whether successful, unsuccessful, or canceled.

Provide documentation about your app to the MobileIron server administrator

Whether your app is an in-house app or is available from the Apple App Store, a MobileIron server administrator configures the server with information about your app. Provide the server administrator documentation that specifies:

- whether you app enforces the print policy.

  The server administrator needs to know whether allowing or not allowing your app to use print capabilities has impact on your app's behavior.

Because the AppConnect library enforces the pasteboard and Open In policies, the server administrator needs no documentation from your app about how it handles it, even if you disable or enable special related user interfaces.

- whether your app handles the pasteboard policy.

  Although the AppConnect library enforces the pasteboard policy, inform the server administrator if your app enables or disables any special user interfaces depending on the policy status. This documentation allows the administrator to better understand your app's expected behavior.

- whether you app handles the Open In policy.

  Although the AppConnect library enforces the Open In policy, provide information so that the server administrator understands your app's expected behavior and recommendations. Specifically, document the following:

  - Whether your app enables or disables any special user interfaces depending on the policy status.
  - Whether your app informs end users when they tap to open a document in an app for which Open In is not allowed. That is, document whether you have implemented the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method.
  - Whether you have a recommended list of whitelisted apps. If you do, document their bundle IDs.
- whether you app handles the Open From policy.

  Although the AppConnect library enforces the Open From policy, provide information so that the server administrator understands your app's expected behavior and recommendations. Specifically, document the following:
  - Whether your app has any special user interfaces depending on the policy status.
  - Whether your app informs end users when they have tapped another app to open a document in your app, but your app is not allowed to receive documents from the other app. That is, document whether you have implemented the `-appConnect:openFromAttemptedWhenACOpenFromPolicyBlocked:` callback method.
  - Whether you have a recommended list of whitelisted apps. If you do, document their bundle IDs.
- whether you app enforces the secure file I/O policy.

  The server administrator needs to know whether your app uses secure file I/O for its sensitive data.
- the app-specific configuration key-value pairs.

  Provide a list of the key-value pairs that your app expects to receive through the AppConnect API. Provide each key's default value if it has one. Specify if the value should default to the device's user's LDAP user ID, password, or email address.
- the encryption group Id app-specific configuration key name for shared secure files.

  If your app uses the Secure file I/O API details to share encrypted files with other AppConnect apps, provide the key name of the encryption group Id that your app expects to receive through the AppConnect API. Also, list the AppConnect apps that your app expects to share files with, so the server administrator can provide the same value to the encryption group Id key for each of those apps.
- the values for the app-specific configuration keys MI_AC_SHARED_GROUP_ID and MI_AC_ACCESS_CONTROL_ID

  If your app provides an extension to share secure files with other AppConnect apps, provide the value of these keys. Your app receives these key-value pairs through the AppConnect API. Also, list the AppConnect apps that your extension expects to share files with, so the server administrator can provide the same key-value pairs for each of those apps.
- AppTunnel information

  If your app expects to interact with internal servers using AppTunnel, specify whether your app expects to work with AppConnect with HTTP/S tunneling, or whether it requires AppConnect with TCP tunneling.

  Also, provide information about the internal servers.

  For example:

  - Explain the type of servers your app interacts with, such as, for example, SharePoint servers.
  - Specify if your app expects to receive internal servers' host names using the app-specific configuration API.
  - Specify if your app expects to be able to interact with all internal servers.
  - If you are an in-house app developer, provide the host names of the internal servers that your app interacts with. Also, provide the port number on each internal server that the app connects to.
- HTTPS connections that your app makes that use certificate authentication to an enterprise service.

  For in-house app developers, provide the URLs of the enterprise services that use certificate authentication.

  If your app receives these URLs through app-specific configuration, make sure you listed the URLs in the app-specific configuration key-value pair documentation.
- Dual-mode app behavior.
  - Provide expected behavior and features in AppConnect mode versus non-AppConnect mode.
  - If your app allows the device user to switch between AppConnect mode and non-AppConnect mode, document what the device user must do.
- Whether your app uses the AppConnect-provided screen blurring capability

  Server administrators need to know whether your app will be impacted if they disable screen blurring for your app.
- Whether your app does not allow some or all custom keyboards.
- Whether your app includes the MI_AC_DISABLE_SCHEME_BLOCKING key set to `YES` in its Info.plist.

AppConnect Library Log Messages

The AppConnect library logs information messages, warnings, and errors. Use these log entries combined with your app's own log entries to debug your app and its use of the SDK.

All AppConnect library log entries begin with:

```
AppConnect:<Log level>
```

Informational log messages

The AppConnect Library logs the following information messages:

- `@"[AppConnect:Status] Starting. Library version: %@"`
  Logged when the app calls `-startWithLaunchOptions:`. The value of `%@` is the version of the AppConnect library.
- `@"[AppConnect:Status] Checkin interval is unknown; attempting checkin.`
  Logged when the app runs for the first time, and the AppConnect library is about to contact the the MobileIron client app.
- `@"[AppConnect:Status] Checkin time is in the past; attempting checkin."`
  Logged when the checkin interval has expired, and the library is about to contact the MobileIron client app.
- `@"[AppConnect:Status] User was inactive; triggering passcode challenge."`
  Logged when the AppConnect passcode auto-lock timeout has expired due to no activity in any AppConnect app. The AppConnect library is about to contact the MobileIron client app to prompt the user to enter the AppConnect passcode.
- `@"[AppConnect:Status] Secure services are now available."`
  Logged when the AppConnect library has received the encryption key from the MobileIron client app, making secure services become available.
- `@"[AppConnect:Status] Secure services are now unavailable."`
  Logged when secure services become unavailable. This message is logged, for example, when the AppConnect passcode's auto-lock timeout expires.
- `@"[AppConnect:Status] App is authorized but secure services are unavailable; attempting checkin."`
  Logged when an app becomes authorized, but the AppConnect library has not yet received the encryption key from the MobileIron client app. The AppConnect library will check in with the MobileIron client app to get the key.
- `@"[AppConnect:Status] Stopping."`
  Logged when the app calls `-stop`.

API usage errors and warnings

The AppConnect Library logs the following errors and warnings when the app has incorrectly called an API:

- `@"[AppConnect:Error] AppConnect cannot be instantiated directly. Instead, call +initWithDelegate: and then +sharedInstance."`
  The app called `-init` on an AppConnect instance, which is not allowed. Instead, call the static method `+initWithDelegate:` of the AppConnect class once. Then use the AppConnect class method `+sharedInstance` to get a reference to the AppConnect singleton.
- `@"AppConnect error: +initWithDelegate: appConnectDelegate must not be nil."`
  The app called `+initWithDelegate:` with a `nil` appConnectDelegate parameter. Provide as the parameter value an instance of the class that conforms to the AppConnectDelegate protocol.
- `@"[AppConnect:Error] +initWithDelegate: has already been called. +initWithDelegate: should only be called once per app launch."`
  The app called `+initWithDelegate:` more than once.
- `@"[AppConnect:Error] Application called -authStateApplied:message: with ACPOLICY_UNSUPPORTED. All applications must support all authStates."`
  Call `-authStateApplied:message:` with its `ACPolicyState` parameter set to either `ACPOLICY_APPLIED` or `ACPOLICY_ERROR`.
- `@"[AppConnect:Warning] Attempted to set policy state for a policy that isn't present, type = %i."`
  This warning is unlikely to occur. The app called one of the notification acknowledgment methods, such as `-pasteboardPolicyApplied:message:`, on a policy that the AppConnect library has not received from the MobileIron client app. The policy type `%i` is a value that the AppConnect library uses internally.
- `@"[AppConnect:Error] AppConnect is unable to start because [UIApplication sharedApplication] is not an instance AppConnectUIApplication."`
  The call in main.m to the function `UIApplicationMain` is incorrect. Follow the instructions in Use AppConnect's UIApplication subclass.

Miscellaneous errors and warning

The AppConnect library logs the following miscellaneous errors and warnings:

- `@"[AppConnect:Error] Invalid %@: URL."`
  The AppConnect library received an ac<*bundleid*>: URL, but the URL was invalid. The AppConnect library discards the URL. The value of `%@` is the invalid URL.
  If you are having issues using the AppConnect library, report these errors to MobileIron Technical Support.
- `@"[AppConnect:Error] internal error"`
  If you are having issues using the AppConnect library, report any errors that begin with `@"AppConnect internal error"` to MobileIron Technical Support.

Developing AppConnect Apps with Xamarin

- Overview of using AppConnect with Xamarin apps
- Available C# bindings
- Xamarin AppConnect sample apps
- How to include the Xamarin C# binding in your Xamarin project

Overview of using AppConnect with Xamarin apps

The AppConnect for iOS SDK provides a Xamarin C# binding for the AppConnect library APIs. This binding allows you to develop iOS AppConnect apps using the Xamarin development platform.

If your AppConnect app is to be distributed from the Apple App Store, due to Apple App Store requirements, your app is required to work as either an AppConnect app or a regular app. See Developing Third-party Dual-mode Apps.

The Xamarin AppConnect C# binding, sample apps, and C# API documentation are available at these sites:

- https://developer.mobileiron.com in appconnect-ios-xamarin-plugin<*version*>_<*build*>.zip
- https://support.mobileiron.com/support/CDL.html in the plugins/xamarin folder of the AppConnectiOSSDK_V<*version*>_<*build*>.zip

The xamarin folder in these ZIP files contains:

- AppConnectSDKBinding.dll
- Docs folder
  Contains the Monodoc documentation of Xamarin AppConnect C# APIs.
- Docs-html folder
  Contains the HTML documentation of Xamarin AppConnect C# APIs, generated for convenience from the Monodoc documentation.
- Samples folder
  Contains the sample apps HelloAppConnectXamarin and DualMode.

For general information about AppConnect, see Introducing the MobileIron AppConnect for iOS SDK.

Available C# bindings

The Xamarin AppConnect C# binding supports all the Objective-C APIs available in the AppConnect library ***with the following exceptions***:

- APIs relating to getting upload status for tunneled HTTP/S requests
- Secure file I/O POSIX-style and Objective-C APIs
- The ACSensitiveData and ACSensitiveMutableData APIs
- The custom cryptography methods `-derivedAppKeyWithIdentifier:error:` and `-derivedSharedKeyWithIdentifier:error:`
- The `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method
- The `-appConnectAttemptedDragAndDropToNonAppConnectApp:` callback method
- The APIs relating to sharing secure files from an extension
- The APIs relating to the Open From policy (Note that the AppConnect library enforces the Open From policy)

The AppConnect C# binding provides documentation for each method, property and enumeration in HTML and in Monodoc format. You can also refer to the information in the Objective-C API descriptions in AppConnect for iOS API.

Xamarin AppConnect sample apps

The AppConnectiOSSDK_V<*version*>_<*build*>.zip contains sample apps that illustrate how to use the Xamarin AppConnect C# binding.

These sample apps are:

- HelloAppConnectXamarin
  This sample app demonstrates how an app uses the Xamarin AppConnect C# binding. The app displays its authorization status, its app configuration, and its data loss prevention policies.
- DualMode
  This sample app demonstrates the behavior of a dual-mode app.
  For an overview of dual-mode apps, see Developing Third-party Dual-mode Apps.

The Xamarin AppConnect C# binding does not provide bindings for the AppConnect secure file I/O APIs. However, it does provide bindings for the APIs that obtain an encryption key for use with custom cryptography routines. Only Xamarin dual-mode apps that use custom cryptographic routines need to keep track of the dual-mode data encryption states that are described in the dual-mode app section.

How to include the Xamarin C# binding in your Xamarin project

The Xamarin AppConnect C# binding is available in AppConnectiOSSDK_V<*version*>_<*build*>.zip in AppConnectSDKBinding.dll.

To include AppConnectSDKBinding.dll in your Xamarin solution using Xamarin Studio:

1. Unzip AppConnectiOSSDK_V<*version*>_<*build*>.zip on to your computer.
2. Open your app's solution in Xamarin Studio.
3. In the iOS project, select **References > Edit References...**
4. Select the **.Net Assembly** tab**.**
5. Click **Browse** to navigate to and select the AppConnectSDKBinding.dll in the unzipped AppConnect SDK folders.
6. Click **Open** to select the DLL file.
7. Click **OK**.

The classes, methods, and properties of the Xamarin AppConnect C# APIs are now available for your app to use.

How to initialize your Xamarin app to use AppConnect C# APIs

To use the AppConnect C# APIs, do the following:

Register as a handler of the AppConnect URL scheme

Your app must handle the AppConnect URL scheme. The MobileIron client app uses this URL scheme to communicate with your app's instance of the AppConnect library.

Register the AppConnect URL scheme by modifying the app's Info.plist. You edit the key called URL types as follows:

1. Set URL Identifier to the app's bundle ID.
   For example:
   `com.mobileiron.ios.xamarin.HelloAppConnect`
2. Set URL Schemes to the app's bundle ID, prefixed with `ac`.
   For example:
   `accom.mobileiron.ios.xamarin.HelloAppConnect`

For example, to edit Info.plist using Xamarin Studio:

1. Open your app's Xamarin solution.
2. Open the app's Info.plist in the property list editor.
3. Select **Advanced**.
4. Click **Add URL Type**.
5. Set URL Identifier to the app's bundle ID.
   For example:
   `com.mobileiron.ios.xamarin.HelloAppConnect`
6. Set URL Schemes to the app's bundle ID, prefixed with `ac`.
   For example:
   `accom.mobileiron.ios.xamarin.HelloAppConnect`

Declare the AppConnect URL scheme as allowed

Declare the `appconnect` and the `alt-appconnect`URL schemes in your app's Info.plist as allowed URL schemes. Your app's instance of the AppConnect library:

- uses the `appconnect` URL scheme to communicate with Mobile@Work or MobileIron Go.
- uses the `alt-appconnect` URL scheme to communicate with MobileIron AppStation.

To allow the `appconnect` and `alt-appconnect` URL schemes, add a key called `LSApplicationQueriesSchemes` to the app's Info.plist as follows:

1. Add a key of type Array.
2. Set the name of the key to `LSApplicationQueriesSchemes`.
3. Add an item to the array.
4. Set the value of the item to `appconnect`.
5. Add another item to the array.
6. Set the value of the item to `alt-appconnect`.

Example : Editing the Info.plist using Xamarin Studio

1. Open your app's Xamarin solution.
2. Open the app's Info.plist in the property list editor.
3. Select **Source**.
4. Select **Add new entry**.
5. Select the **+**.
6. Change the name of the property from **Custom Property** to **LSApplicationQueriesSchemes**.
7. In the **Type** column, select **Array**.
8. Select **Add new entry**, which appears indented under the new property.
9. Select the **+**.
10. In the **Value** column for the new **String** item, enter **appconnect**.
11. Similarly, add a new entry to the **LSApplicationQueriesSchemes** array with the value **alt-appconnect**.

Add AppConnect-related entries to your Info.plist

Enable screen blurring

The AppConnect library can automatically blur your app's screen whenever it is not active. This security measure protects the app's data from being captured in screenshots. The AppConnect library blurs the screen when `-applicationWillResignActive:` is called and unblurs it when `-applicationDidBecomeActive:` is called.

To enable screen blurring, add the key MI_AC_PROVIDE_SCREEN_BLUR to your app's Info.plist as a Boolean. Set the value to YES.

When you set the Info.plist key MI_AC_PROVIDE_SCREEN_BLUR to YES, the MobileIron server administrators can disable screen blurring by setting a key-value pair on the server for your app's configuration. The server key is MI_AC_ENABLE_SCREEN_BLURRING with the value false.

NOTE: If you already implemented screen blurring in your app, remove that code and use the MI_AC_PROVIDE_SCREEN_BLUR plist key. Using the plist key ensures that all AppConnect apps behave consistently.

Allow Face ID

Include **Privacy - Face ID Usage Description** to your app's info.plist, with a string value indicating the purpose of Face ID use. For example, add the value **AppConnect**. If you manually add this key, its name is NSFaceIDUsageDescription.

Server administrators can allow the use of Touch ID or Face ID instead of an AppConnect passcode. Therefore, this Info.plist entry is required on iOS 11 through the most recently released version as supported by MobileIron.

Use AppConnect's UIApplication subclass

To use AppConnect's UIApplication subclass:

1. Open Main.cs for editing.
2. Change the second argument of the call to `UIApplication.Main()` to `AppConnectBinding.Constants.kACUIApplicationClassName`.
   The second argument, the `principalClassName` argument, is the UIApplication class or subclass for the app.
3. Make sure the third argument of the call to `UIApplication.Main()` is your UIApplicationDelegate subclass name.

For example, in the HelloAppConnectXamarin app provided with the AppConnect for iOS SDK, the statement that calls `UIApplication.Main()` is:

```
UIApplication.Main(args, AppConnectBinding.Constants.kACUIApplicationClassName,
                   "HACAppDelegate"));
```

NOTE: If you use a subclass of UIApplication for your app:

1. Derive your subclass from AppConnectUIApplication instead of UIApplication.
2. Use the name of your AppConnectUIApplication subclass for the `principalClassName` argument in the call to `UIApplication.Main()`.
3. When you override a UIApplication method in your AppConnectUIApplication subclass, always invoke the method implementation of the superclass AppConnectUIApplication at the end of your method.
   If you do not invoke the superclass implementation, AppConnect features will not work in your app.

Initialize the AppConnect library

To initialize the AppConnect library for your app to use:

Edit your AppDelegate source file

1. Open your AppDelegate source file for editing.
2. Add the following line to your `using` statements:
   ```
   using AppConnectBinding;
   ```

Create a subclass of AppConnectDelegate

In your AppDelegate source file:

1. Create a subclass of AppConnectDelegate. Do the following:
   - Implement each abstract method in AppConnectDelegate.
   - Implement each virtual method in AppConnectDelegate that your app's functionality requires.

   For example, in HelloAppConnectXamarin, in AppDelegate.cs, the HACAppConnectDelegate class derives from the AppConnectDelegate class.
   Details about each method is available in the code. You can also refer to the corresponding Objective-C method in AppConnect for iOS API.
2. If you want to retrieve you app's original UIApplicationDelegate object, model your code from this line from HelloAppConnectXamarin:

   ```
   this.hacAppDelegate = (HACAppDelegate)
      ((AppConnectUIApplication)UIApplication.SharedApplication).OriginalDelegate;
   ```

NOTE: Xamarin apps must use `OriginalDelegate` to get the UIApplicationDelegate object. For more information, see originalDelegate property (deprecated).

Modify your UIApplicationDelegate subclass

Modify your UIApplicationDelegate subclass as follows:

1. Instantiate the AppConnectDelegate subclass object.
   For example, in HelloAppConnectXamarin:
   ```
   this.appConnectDelegate = new HACAppConnectDelegate();
   ```
2. Call the static method `InitWithDelegate()` of the AppConnect class.The method takes as a parameter an object of the AppConnectDelegate subclass.
   For example, in HelloAppConnectXamarin, in the HACAppDelegate class implementation, the method `FinishedLaunching()` calls `InitWithDelegate()` as follows:
   ```
   AppConnect.InitWithDelegate(this.appConnectDelegate);
   ```
3. Save the singleton instance of the AppConnect library.
   For example, in HelloAppConnectXamarin, the HACAppDelegate object saves the singleton instance in the `appConnect` member variable:
   ```
   this.appConnect = AppConnect.SharedInstance;
   ```
4. Call the AppConnect singleton's method `StartWithLaunchOptions()`.
   The app must:
   - Call this method from its AppDelegate's method `FinishedLaunching()`
   - Pass along its `options` parameter value.

   For example, in HelloAppConnectXamarin:
   ```
   this.appConnect.StartWithLaunchOptions(options);
   ```
   After this step, the AppConnect singleton is initializing. However, the app cannot yet use the singleton's *instance* properties. The app can:
   - use the AppConnect *class* properties.
   - use the methods of the AppConnect singleton object.
5. If your application supports UIScene, call the AppConnect singleton's method `SceneWillConnectToSessionWithOptions()` from your UIScene delegate's `void WillConnect (UIScene scene, UISceneSession session, UISceneConnectionOptions connectionOptions)` method passing `connectionOptions` as an input parameter.
   For example:
   ```
   public class MySceneDelegate : UIWindowSceneDelegate {
          public override void WillConnect (UIScene scene, UISceneSession session,
   UISceneConnectionOptions connectionOptions)
          {
                 AppConnect.SharedInstance.SceneWillConnectToSessionWithOptions(connectionOptions)
          }
   }
   ```
6. Indicate in the user interface that the app is initializing if the app requires the AppConnect singleton's instance properties to determine what to do. For example, use an activity indicator (spinner). Remove the

indication after the app is notified that the AppConnect singleton is ready.
One reason this indication is important involves when to display sensitive data. Do not show any sensitive data until the AppConnect singleton is ready, because until that time, the app cannot determine whether it is authorized. Only an authorized app should show sensitive data.

Wait for the AppConnect singleton to be ready

The app cannot use the AppConnect singleton's instance properties until the `Ready` property on the AppConnect singleton is set to `true`. It is set to `true` when the callback method `AppConnectIsReady()` in your AppConnectDelegate subclass is called. The app can now access the instance properties, such as `AuthState` and `PasteboardPolicy`, on the AppConnect singleton.

Before accessing any instance properties, use the `Ready` getter to make sure the properties are accessible.

For example, in HelloAppConnectXamarin, the `AppConnectIsReady()` callback method calls `UpdateLabels()`. The `UpdateLabels()` method calls various methods that access the instance properties on the AppConnect singleton. Because other methods also call `UpdateLabels()`, `UpdateLabels()` first checks the `Ready` property:

```
if (this.appConnect.Ready) {

    // Call methods that access instance properties.
}
else {
        authInfoText = "Ready: NO (AppConnect is not ready yet)";
        policyInfoText = "AppConnect is not ready yet";
        configInfoText = "AppConnect is not ready yet";
}
```

For details about the `AppConnectIsReady()` callback method andthe `Ready` property, see the code. You can also refer to the corresponding Objective-C information in AppConnect ready API details .

Optional: Specify app permissions and configurations in a plist file

If your app is an in-house app, you can specify default values for:

- the data loss prevention policies, such as the Open In policy
- the key-value pairs for your app-specific configuration

Specifically, you can provide a special plist file called AppConnect.plist as part of your in-house app that:

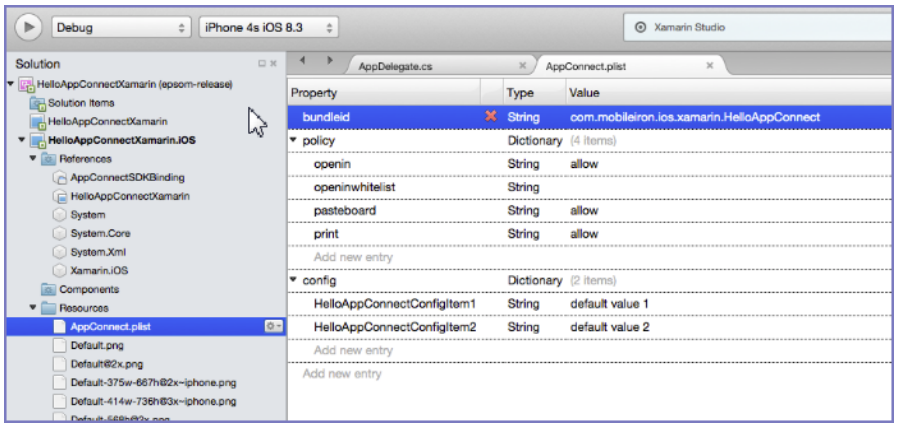
- specifies whether your app should be allowed by default to copy to the iOS pasteboard, use document interaction (Open In and Open From), and print.
- specifies app-specific configuration keys and default values.

These default values are used by MobileIron server to make it easier for the server administrator to set up your app with the correct data loss prevention policies and app-specific configurations. *Your app never reads the AppConnect.plist*.

When you include the AppConnect.plist in your app:

1. When an administrator uploads your in-house app to the MobileIron server, the server uses this plist file to automatically create server policies that contain your specified data loss prevention policies and app-specific configuration.
2. The administrator can then edit these policies.
   For example:
   - If one of your app-specific configuration keys requires a URL of an enterprise server, the administrator provides that value.
   - If the administrator requires stricter data loss prevention policies than your app's default values, the administrator changes the values.
3. The administrator then applies these policies to the appropriate set of devices.
4. When your app runs, it receives the data loss prevention policies and app-specific configuration by using the AppConnect for iOS APIs.
   For example, to handle app-specific configurations, you use the `Config` property (an `NSDictionary` object) and the callback method `ConfigChanged()`.
5. If the administrator later changes the data loss prevention policies or app-specific configuration, your app receives the updates by using the AppConnect for iOS APIs.

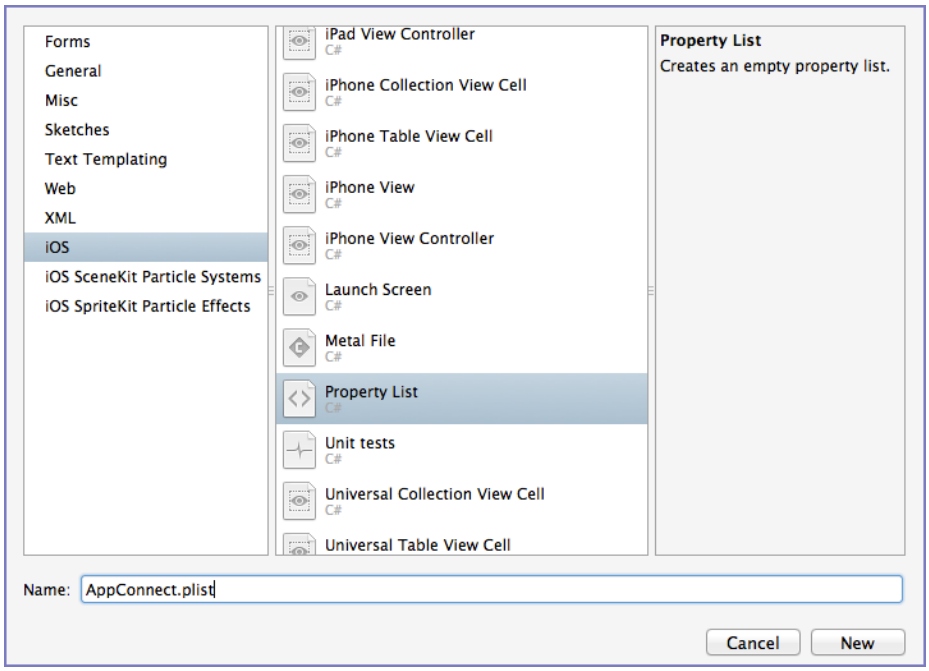
An example of an AppConnect.plist file as viewed in Xamarin Studio looks like the following:



Create the AppConnect.plist in Xamarin Studio

To create an AppConnect.plist file in your Xamarin solution using Xamarin Studio:

1. Open your app's Xamarin solution.
2. Select **Resources > Add New File.**



3. Select **iOS.**
4. Select **Property List.**
5. For **Name**, enter AppConnect.plist.
6. Click **New**.

Edit the AppConnect.plist

1. In the Root key of AppConnect.plist, place a key called `bundleid` with the type String, and set the value to the bundle ID of your app.
2. In the Root key of AppConnect.plist, create two keys called `policy` and `config`, each with the type Dictionary.
3. In the `policy` dictionary, create keys called `openin`, `openinwhitelist`, `openfrom`, `openfromwhitelistpasteboard`, and `print`, each with the type String.
4. Set these keys' values as given in the following table:

TABLE 41. APPCONNECT.PLIST KEYS AND VALUES

Key	Possible values and meanings
`openin`	• `allow` Document interaction is allowed with all other apps. • `disable` Document interaction is not allowed. • `whitelist` Only documents in the `openinwhitelist` list can open documents from your app. • `appconnect` Document interaction is allowed with all other AppConnect apps. NOTE: This value results in the app receiving a whitelist in the Open In policy API. The whitelist contains the list of all currently authorized AppConnect apps. You do not enter an `openinwhitelist` key in the plist. See The openInPolicy and openInWhitelist properties .
`openinwhitelist`	Semi-colon separated list of the bundle IDs of the apps with which document interaction is allowed. This key is necessary when the `openin` key has the value `whitelist`.
`pasteboard`	• `allow` Pasteboard interaction is allowed with all other apps. That is, this option allows the device user to be able to copy content from your app to the iOS pasteboard. Then, any app can copy from the content from the pasteboard. • `disable` Pasteboard interaction is not allowed. • `appconnect` Pasteboard interaction is allowed only with other AppConnect apps. That is, this option allows the device user to be able to copy content from your app to the iOS pasteboard. Then, only other AppConnect apps can copy from the content from the pasteboard.
`print`	• `allow` Printing is allowed. • `disable` Printing is not allowed.

5. In the `config` dictionary, create keys as required for your app.
6. Optionally, add values for the keys. The values must be String types.

The value `$USERID$` in the example tells MobileIron Core to substitute the device user's user ID for the value. Other possible variables for Core are `$EMAIL$` and `$PASSWORD$`. Depending on the Core configuration, custom variables called `$USER_CUSTOM1$` through `$USER_CUSTOM4$` are sometimes available.

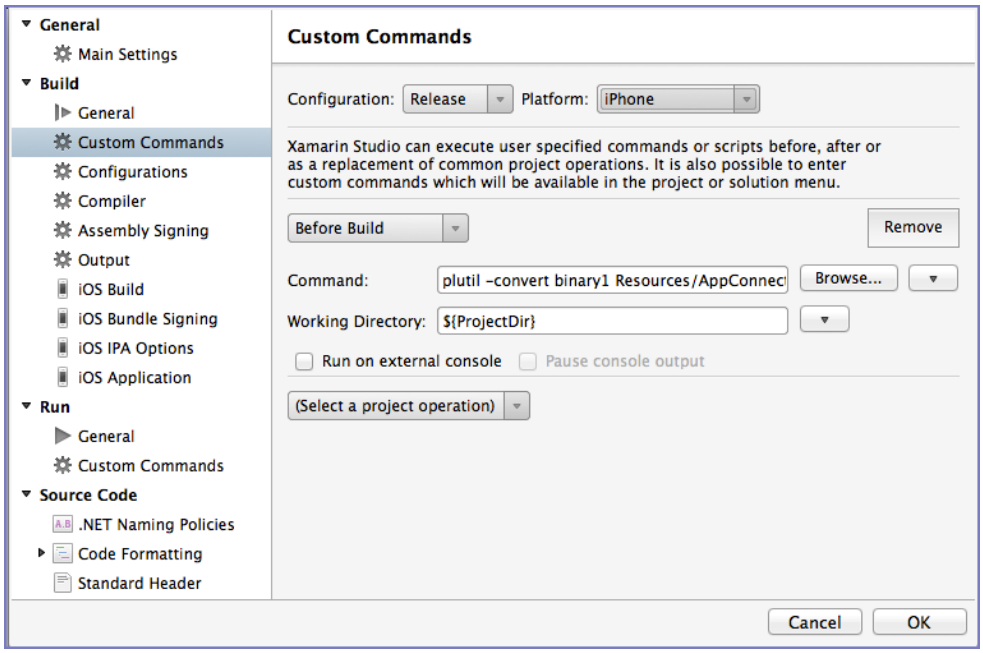
Convert the AppConnect.plist to binary format

The MobileIron server requires that the AppConnect.plist uses binary plist format. When creating an iOS app with Xamarin Studio, you must manually convert the AppConnect.plist to binary format. You can convert

AppConnect.plist using a custom build command.

Do the following:

1. In Xamarin Studio, select your project.
2. Select **Options**.
3. Select **Build > Custom Commands**.



4. For **Configuration**, select **Release**.
5. For **Platform**, select **iPhone**.
6. Select **Before Build**.
7. In the **Command** field, enter:
   `plutil -convert binary1 Resources/AppConnect.plist`
8. Set **Working Directory** to `${ProjectDir}`.
9. Click **OK**.
10. Repeat steps, this time selecting **Debug** for the **Configuration** field.

AppTunnel support in Xamarin apps

Apps built with the Xamarin development platform can access network servers various ways. AppTunnel with HTTP/S tunneling is supported only as follows:

- The app uses the NSURLConnection or NSURLSession APIs exposed to C# through the Xamarin.iOS binding.
- The app uses the ModernHttpClient library with NSURLSession. The ModernHttpClient library with CFNetwork will not work.
  For example, the app initializes the instance of the ModernHttpClient as follows:

```
var httpClient = new HttpClient (new NativeMessageHandler ());
```

AppTunnel Diagnostic API for Xamarin

The AppTunnel Diagnostic API for Xamarin provides troubleshooting information for an app's use of AppTunnel with HTTP/S tunneling. Typically, you add a user interface, such as a menu item, to invoke a diagnostic run for tunneling to a specified URL. Your app then displays or logs the results of the diagnostic run. The API performs the following diagnostics:

TABLE 42. DIAGNOSTICS PERFORMED BY APPTUNNEL DIAGNOSTIC API FOR XAMARIN

Diagnostic	Description
Run life cycle	Tests the beginning, ending, and restarting of connections. Redirects restart the connection with a new URL, new cookies, and/or new connection settings.
Policy integrity	Checks that the following elements in the AppTunnel policy that relate to the request are valid: • Client identity • Server certificate • At least one tunneling rule in the policy • A rule that matches the request
Certificate challenges	Evaluates the certificate from the sentry, and uses the client identity to authenticate with the server. If both of these challenges succeed, the API establishes a connection with the sentry. If you start another run while the connection is still established, the new run will not perform any certificate related diagnostics.
Connection results	Presents the data received by the app from the backend server.

Set up your app to use the AppTunnel Diagnostic API for Xamarin

See AppTunnel Diagnostic API for Xamarin for instructions on setting up your app to use this API.

Run the API

This API is a copy of the native AppTunnel diagnostic API, with the exception that C# nomenclature is used. For details on running this API, please refer to the AppConnect C# binding which provides documentation for each method, property and enumeration in HTML and in Monodoc format.

API Response

The API returns the following series of messages to the console:

TABLE 43. APPTUNNEL DIAGNOSTIC API RESPONSE MESSAGES

Message	Message Content	Description
1	Success: Diagnostic run started. Requesting (URL) Failure: N/A	Indicates successful start of the API run.
2	Success: Request matches a tunneling rule so it will be tunneled. Failure: • Request does not match a tunneling rule so it will not be tunneled • AppTunnel policy has no tunneling rules.	Succeeds if an initial or redirected request matched a tunneling rule, or fails otherwise.
3	Success: Server certificate in the AppTunnel policy is valid. Failure: • No server certificate was found in the AppTunnel policy. • Server certificate in the AppTunnel policy is invalid. It may have expired.	Succeeds if the policy contains a valid server certificate, or fails otherwise.
4	Success: Server certificate passed all evaluation Failure: Server certificate was not trusted. The trust result was (trust result)	Succeeds if the sentry's server-side certificate is valid, or fails otherwise.
5	Success: Failure: • Server issued an auth challenge type that the diagnostic does not support. • Aborting the diagnostic and the auth challenge. Auth challenge type is (auth type)	Returns message if the diagnostic is aborted because the server issued an auth challenge that the diagnostic does not support. Returns no message on success.
6	Success: Client identity in the AppTunnel policy appears to be valid. Failure: • No client identity in the AppTunnel policy.	Succeeds if the policy contains a valid client identity, or fails otherwise.

TABLE 43. APPTUNNEL DIAGNOSTIC API RESPONSE MESSAGES (CONT.)

Message	Message Content	Description
	• Client certificate in the AppTunnel policy is invalid. It may have expired.	
7	Success: Authenticated with client identity Failure: There was a previous failure of the client auth challenge.	Succeeds if the client-side certificate was sent, or fails otherwise.
8	Success: The server redirected to a new URL. Redirected by server to new URL (url) Failure: N/A	Always succeeds.
9	Success: Received HTTP status code (1xx, 2xx, or 3xx) Failure: Received HTTP status code (4xx or 5xx)	The server returns an HTTP status code. Status codes in the 1xx, 2xx, and 3xx range indicate success. Status codes in the 4xx and 5xx range indicate failure.
10	Success: Received (bytes) bytes of data Failure: No message appears	If data is received, the API returns a message.
11	Success: Session completed normally Failure: Session completed with error: (error)	Fails if the session completed with an error, or succeeds otherwise.

Sample response

```
ACTDR: [1] Success: Diagnostic run started. Requesting http://httpstat.us/200
ACTDR: [1] Success: Request matches a tunneling rule so it will be tunneled.
ACTDR: [1] Success: DNS resolution of the Sentry app081.auto.mobileiron.com succeeded
ACTDR: [1] Success: Server certificate in the AppTunnel policy is valid.
ACTDR: [1] Success: Server certificate passed all evaluation
ACTDR: [1] Success: Client identity in the AppTunnel policy appears to be valid.
ACTDR: [1] Success: Authenticated with client identity
ACTDR: [1] Success: Received HTTP status code 200
ACTDR: [1] Success: Received 6 bytes of data
ACTDR: [1] Success: Session completed normally
ACTDR: [1] Ended
```

FIPS Compliance in an AppConnect SDK App

You can make an AppConnect app FIPS compliant. FIPS compliance information is available at:

http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

The following features of the AppConnect for iOS SDK allow you to make a FIPS compliant AppConnect app:

- The SDK is FIPS compliant on all iOS devices running supported versions of iOS as listed in Product versions required .
  It is not FIPS compliant on devices running previous iOS versions.

The AppConnect for iOS SDK uses ECDH and AES-256-GCM protocols for the inter-app communication bus between AppConnect apps and Mobile@Work.

- The SDK uses FIPS compliant algorithms for all cryptographic operations.
- The SDK uses OpenSSL for cryptography.
  The use of OpenSSL allows you to link into a FIPS compliant version of the OpenSSL library in your app.

To make your app is FIPS compliant *with regard to its use of the AppConnect for iOS SDK*, do the following:

- Link into an OpenSSL library built in FIPS mode. When you link your OpenSSL library to your Xcode project, make sure it is listed ***higher than*** the AppConnect.framework in Xcode under **Linked Frameworks and Libraries**.

MobileIron has verified that the AppConnect for iOS SDK works correctly using OpenSSL library version 1.0.2h. Check OpenSSL documentation to determine differences with other OpenSSL library versions.

- Make sure that you have initialized OpenSSL in FIPS mode before calling any AppConnect for iOS APIs.
- If you use your own libcrypto.a file, make sure it is FIPS compliant. The libcrypto.a file included in the AppConnect.framework is FIPS compatible.

Testing for Third-party App Developers

Third-party AppConnect app testing overview

Test your app using the instructions in this chapter or the instructions in Testing for In-house App Developers based on the following table:

TABLE 44. WHERE TO FIND THE RIGHT TESTING INSTRUCTIONS

Your role	Testing instructions
Third-party app developer	This chapter
In-house app developer whose organization uses MobileIron Cloud	This chapter
In-house app developer whose organization uses MobileIron Core or Connected Cloud.	See Testing for In-house App Developers.

Testing with MobileIron Core as described in this chapter is necessary to verify the AppConnect-related functionality of your AppConnect app. If your app accesses servers behind a firewall using AppTunnel, a Standalone Sentry is necessary to verify the AppTunnel feature. All AppConnect apps require Mobile@Work to interact with Core.

For testing your app, MobileIron provides you access to MobileIron Connected Cloud, the cloud offering of the on-premise server MobileIron Core. MobileIron also provides you access to Standalone Sentry if necessary. You then use a web portal called the Admin Portal to make configuration changes necessary for testing your app.

NOTE: Apps that you test with MobileIron Connected Cloud and Mobile@Work will also work with MobileIron Cloud and supported versions of MobileIron Go. However, some AppConnect features are not supported by MobileIron Cloud and MobileIron Go.

Use an enterprise build of your app for testing. When your app is completely tested, build a distribution build for distributing the app through the Apple App Store. These procedures are for testing only.

Before you begin:

- Contact MobileIron to provide you with a Core (Connected Cloud) and (if necessary) Standalone Sentry.
- Get Mobile@Work from the Apple App Store.

Set up MobileIron Core

To set up Core for testing your AppConnect app, do the following high-level steps:

1. Login to the Admin Portal.
2. Enable AppConnect on MobileIron Core.
3. Configure the AppConnect global policy.
4. Create an AppConnect container policy.

NOTE: These instructions are for Core 9.7.0.0.

Login to the Admin Portal

MobileIron provides you with the following information about your test MobileIron Core:

- the URL for accessing the Core's Admin Portal
  The Admin Portal is a web portal for configuring Core. The URL has the format:
  https://m.mobileiron.net/<*app partner name*>
- a user ID and password for accessing the Admin Portal
  You also use this user ID to register a device with Core.
- a port number for Core, used when you register a device with Core.
  The port number is typically four or five digits.

To login to Core:

1. Open a browser to the URL for accessing the Core's Admin Portal.
   Use the URL of your test Core, appended with /mifs. For example:
   https://m.mobileiron.net/myCompany/mifs
2. Enter your Username and Password.
3. Click Sign In.
   You are now in the Admin Portal.
   Change your password when prompted.

Enable AppConnect on MobileIron Core

To enable AppConnect on Core:

1. In the Admin Portal, go to Settings.
2. Select Additional Products > Licensed Products.

3. Select AppConnect For Third-party And In-house Apps if it is not already selected.
4. Click Save.

Configure the AppConnect global policy

An AppConnect global policy is necessary for your AppConnect app to work properly.

To configure an AppConnect global policy:

1. In the Admin Portal, select Policies & Configs > Policies.
2. Select the row that says Default AppConnect Global Policy for the Policy Name.
3. Click Edit in the right-hand pane.
4. For AppConnect, select Enabled.
   The display now shows all the AppConnect global policy fields.
5. In the AppConnect Passcode section, for Passcode Type, select Numeric.
6. In the AppConnect Passcode section, select Passcode Is Required For iOS Devices.
7. Click Save.

NOTE: Do not select Authorize in the field Apps Without An AppConnect Container Policy in the section Data Loss Prevention Policies in the AppConnect global policy. You will authorize the app with an AppConnect container policy instead.

Create an AppConnect container policy

An app is authorized only if an AppConnect container policy for the app is present on the device.

To create an AppConnect container policy:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select Add New > AppConnect > Container Policy.
3. Enter a name for the AppConnect container policy.
   For example: My App's Container Policy
4. In the Application field, enter the bundle ID of your app.
   For example: com.MyCompany.MySecureApp
5. Click Save.
   The dialog box closes and the new AppConnect container policy appears in the list.
6. Select the AppConnect container policy you just created.
7. Select Actions > Apply To Label.
8. Select iOS.
9. Click Apply.
10. Click OK.

Set up your end-user device

To set up your end-user device, do the following high-level steps:

1. Set up Mobile@Work on an iOS device.
2. Install your app on the device.
3. Set up the AppConnect passcode on the device.

Set up Mobile@Work on an iOS device

To set up Mobile@Work for iOS on your device:

1. Download and install Mobile@Work from the Apple App Store.
2. Tap the MobileIron app icon to launch Mobile@Work.
3. Enter the user name that MobileIron gave you.
   You use the same user name that you use to log into the Admin Portal.
4. Enter the server as follows:
   m.mobileiron.net:<*port number*>
   where <*port number*> is the port number you received from MobileIron along with your user name and password.
   For example:
   m.mobileiron.net:27643
5. Enter the password.
   Enter the password that you created when you first logged into the Admin Portal.
6. Follow the prompts from Mobile@Work to complete its setup.
   Allow Mobile@Work to use the current location.
   Install new profiles and certificates when prompted.

Install your app on the device

Install your app on the device in the same way you install any app that you are testing.

Set up the AppConnect passcode on the device

When you run your app for the first time, Mobile@Work prompts you to create the AppConnect passcode. Follow the steps to create the AppConnect passcode.

Test authorization status handling

You can make changes to Core configuration to test your app's handling of the different authorization statuses: authorized, unauthorized, and retired.

Change the status to authorized or unauthorized

A security policy on Core specifies the requirements for a device. If a device is not compliant with a requirement, the security policy specifies a compliance action. One compliance action is to block AppConnect apps on the device, which means that the apps become unauthorized.

The list of requirements that can impact authorization is long, but for testing your app, you need to work with only one requirement. The requirement involves a list of device models that are not allowed to use AppConnect apps.

Therefore, to unauthorize the app on the device:

1. In the Admin Portal, select Policies & Configs > Policies.
2. Select the Default Security Policy.

3. Click Edit in the right-hand pane.
4. Scroll down to the section called Access Control, under For iOS Devices.
5. Select Block Email, AppConnect Apps, And Send Alert For The Following Disallowed Devices.
6. Move the model of your test device to the Disallowed area.
7. Click Save.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is unauthorized. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the change to the unauthorized state. Specifically, verify that your app:

- exits any sensitive part of the application.
- stops allowing the user to access sensitive data and views.
- displays the message received in the callback method that explains the authorization status change.
- calls the `-authStateApplied:message:` method.

To re-authorize the app on the device:

1. In the Admin Portal, select Policies & Configs > Policies.
2. Select the Default Security Policy.
3. Click Edit in the right-hand pane.
4. In the section called Access Control, under For iOS Devices, *uncheck* Block Email, AppConnect Apps, And Send Alert For The Following Disallowed Devices.
5. Click Save.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is authorized. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the change to the authorized state. Specifically, verify that your app:

- allows the user to access sensitive data and views.
- calls the `-authStateApplied:message:` method.

Change the status to retired

An app is authorized only if an AppConnect container policy for the app is present on the device. If you remove the AppConnect container policy from the device, the app becomes retired.

To retire the app on the device:

1. In the Admin Portal, select Policies & Configs > Configurations.

2. Select the AppConnect container policy for your app.
3. Select Actions > Remove From Label.
4. Select iOS.
5. Click Remove.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is retired. Otherwise, it receives the notification the next time it runs. The message string in the notification is the default unauthorized message:
   "Your administrator has not authorized this app."

Verify that your app correctly handles the change to the retired state. Specifically, verify that your app:

- exits any sensitive part of the application.
- deletes all sensitive data, including any stored authentication credentials, data in files, keychain items, pasteboard data, and any other persistent storage.
- displays the message received in the callback method that explains the authorization status change.
- calls the `-authStateApplied:message:` method.

Reauthorize a retired app

A retired app is sometimes re-authorized at a later time.

To reauthorize the retired app on the device:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select the AppConnect container policy for your app.
3. Select Actions > Apply To Label.
4. Select iOS.
5. Click Apply.
6. Click OK.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is authorized. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the change to the authorized state. Specifically, verify that your app:

- dismisses any user interface that displays that the user is not authorized to use the app.
- allows the user to access sensitive data and views.
- calls the `-authStateApplied:message:` method.

Test data loss prevention policy handling

The AppConnect container policy for your app specifies its data loss prevention (DLP) policies. In this policy, you specify whether your app is allowed to:

- copy content to the iOS pasteboard.
- drag and drop content to other apps
- print by using AirPrint, any future iOS printing feature, any current or future third-party libraries or apps that provide printing capabilities.
- share documents with other apps.

By changing the AppConnect container policy, you can test:

- your app's behavior for each data loss prevention policy.
- how your app handles changes to the policies in the notification callback methods in the AppDelegateProtocol.

To change the DLP policies:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select the AppConnect container policy for your app.
3. Click Edit in the right-hand pane.
4. Allow or prohibit features relating to data loss prevention policies as follows:

TABLE 45. DLP POLICY DESCRIPTIONS

DLP policy	Description
Allow Print	Select Allow Print if you want the app to use the device's print capabilities.
Allow Copy/Paste To	Select Allow Copy/Paste To if you want the device user to be able to copy content from the AppConnect app to other apps. When you select this option, then select either: • All Apps Select All Apps if you want the device user to be able to copy content from the AppConnect app and paste it into any other app. • AppConnect Apps Select AppConnect Apps if you want the device user to be able to copy content from the AppConnect app and paste it into only other AppConnect apps.
Allow Drag and Drop	Select Allow Drag and Drop if you want the device user to be able to drag content from the AppConnect app to other apps. When you select this option, then select either: • All Apps Select All Apps if you want the device user to be able to drag content from the AppConnect app to any other app. • AppConnect Apps Select AppConnect Apps if you want the device user to be able to drag content from the AppConnect app to only other AppConnect apps.
Allow Open In	Select Allow Open In if you want the app to be allowed to use the device's Open In

TABLE 45. DLP POLICY DESCRIPTIONS (CONT.)

DLP policy	Description
	(document interaction) feature. When you select this option, then select either: • All Apps Select All Apps if you want the app to be able to send documents to any other app. • AppConnect Apps Select AppConnect Apps to allow an AppConnect app to send documents to only other AppConnect apps. NOTE: This option results in the `openInPolicy` property having the value `ACOPENINPOICY_WHITELIST`. Also, the `openInWhitelist` property contains the list of currently authorized AppConnect apps. • Whitelist Select Whitelist if you want the app to be able to send documents only to the apps that you specify. Enter the bundle ID of each app, one per line, or in a semicolon delimited list. For example: com.myAppCo.myApp1 com.myAppCo.myApp2;com.myAppCo.myApp3 The bundle IDs that you enter are case sensitive.

5. Click Save.
6. Click Yes to confirm.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notifications for the updated DLP policies. Otherwise, it receives the notifications the next time it runs.

Verify that your app correctly handles the data loss prevention policy changes, as shown in the following table:

TABLE 46. WHAT TO VERIFY WHEN A DLP POLICY CHANGES

Policy change	What to verify
Allow copy/paste to for all apps	• Verify that the user can cut or copy text, images, or other data to the pasteboard. • Where appropriate, verify that any special user interface that offers the ability to cut or copy data is available and enabled. Also, verify that your app calls the `-pasteboardPolicyApplied:message:` method.
Allow copy/paste to for AppConnect Apps only	• Verify that the user can cut or copy text, images, or other data to the pasteboard. • Where appropriate, verify that any special user interface that offers the ability to cut or copy data is available and enabled. • Verify that the user can paste the data from the pasteboard only into other AppConnect apps. Also, verify that your app calls the `-pasteboardPolicyApplied:message:` method.
Do not allow copy/paste to	• Verify that the user cannot to cut or copy text, images, or other data to the pasteboard. • Where appropriate, verify that any special user interface that offers the ability to cut or copy data is removed or disabled. • Verify your implementation of the callback method `-appConnect:copyAttemptedWhenUnauthorized:`. Also, verify that your app calls the `-pasteboardPolicyApplied:message:` method.
Allow drag and drop to only AppConnect apps	Verify your implementation of the callback method `-appConnectAttemptedDragAndDropToUnauthorizedApp:`.
Allow open in for all apps	Verify that your app enables user interfaces, if any, that give the user the option to use Open In. Also, verify that your app calls the `-openInPolicyApplied:message:` method.
Allow open in for AppConnect apps	Verify that: • your app enables user interfaces, if any, that give the user the option to use Open In. • your app calls the `-openInPolicyApplied:message:` method. • the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method, if implemented, behaves as you expect.
Allow open in for whitelisted apps	Verify that: • your app enables user interfaces, if any, that give the user the option to use Open In. • your app calls the `-openInPolicyApplied:message:` method. • the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:`

TABLE 46. WHAT TO VERIFY WHEN A DLP POLICY CHANGES (CONT.)

Policy change	What to verify
	callback method, if implemented, behaves as you expect.
Do not allow open in	Verify that: • your app disables user interfaces, if any, that give the user the option to use Open In. • your app calls the `-openInPolicyApplied:message:` method. • the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method, if implemented, behaves as you expect.
Allow print	For each part of your app that allows the user to print secure data, verify the capability is enabled. Also, verify that your app calls the `-printPolicyApplied:message:` method.
Do not allow print	For each part of your app that allows the user to print secure data, verify the capability is removed or disabled. Also, verify that your app calls the `-printPolicyApplied:message:` method.

Test AppConnect configuration change handling

AppConnect app configuration on MobileIron Core specifies key-value pairs for configuring your app. You add, and edit, key-value pairs using the Admin Portal.

By changing the AppConnect app configuration, you can test your app's `-appConnect:configChangedTo:` method in the AppDelegateProtocol.

Create an AppConnect app configuration

To create an AppConnect app configuration:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select Add New > AppConnect > App Configuration.
3. Enter a name for the AppConnect app configuration.
   For example: My App's App Configuration
4. In the Application field, enter the bundle ID of your app.
   For example: com.MyCompany.MySecureApp
5. In the App-specific Configurations section, click Add**+** to add a key-value pair.
6. Enter the key-value pairs.

TABLE 47. KEY-VALUE PAIRS IN AN APPCONNECT APP CONFIGURATIONKEY

Key	The key is any string that the app recognizes as a configurable item. For example: userid, appURL
Value	Enter the value. The value is either: • a string The string can have any value that is meaningful to the app. It can also include one or more of these MobileIron Core variables: $USERID$, $EMAIL$, $USER_CUSTOM1$, $USER_CUSTOM2$, $USER_CUSTOM3$, $USER_CUSTOM4$. If you do not want to provide a value, enter $NULL$. The $NULL$ value tells the app that the app user will need to provide the value. Examples: $USERID$ https://someEnterpriseURL.com • a Certificate Enrollment or Certificates setting Certificate Enrollment and Certificate settings that are configured in Policies & Configs > Configurations appear in the dropdown list. When you choose a Certificate Enrollment or Certificate setting, Core sends the contents of the certificate as the value. The contents are base64-encoded. If the certificate is password-encoded, Core automatically sends another key-value pair. The key's name is the string <*name of key for certificate*>_MI_CERT_PW. The value is the certificate's password.

7. Click Save.
8. Click Yes to confirm.
9. Select the new AppConnect app configuration.
10. Select Actions > Apply To Label.
11. Select iOS.
12. Click Apply.
13. Click OK.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification for the new configuration. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the new configuration, correctly applying and using the configured options according to your app's requirements and design.

Update the AppConnect app configuration

To update the AppConnect app configuration:

1. In the Admin Portal, select Policies & Configs > Configurations.

2. Select the your app's AppConnect app configuration.
3. Click Edit in the right-hand pane.
4. In the App-specific Configurations section, click Add**+** to add a key-value pair. To delete a key-value pair, click the X on the row.
5. Update the key-value pairs as described in Create an AppConnect app configuration.
6. Click Save.
7. Click Yes to confirm.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification for the updated configuration. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the updated configuration, correctly applying and using the configured options according to your app's requirements and design.

Test using AppTunnel

Using MobileIron's AppTunnel feature, your app can securely tunnel HTTP and HTTPS network connections from the app to servers behind an organization's firewall. Your app does not take any special actions related to tunneling; the AppConnect library, Mobile@Work, and a Standalone Sentry handle tunneling for the app.

You can test the HTTP/S tunneling capability using the provided MobileIron Core and Sentry. Using the Admin Portal, you configure app-specific AppTunnel settings for Core and Sentry.

Before you begin: Contact MobileIron to provide you with a Standalone Sentry.

To test your app's use of AppTunnel with HTTP/S tunneling, do these high-level steps:

1. Enable AppTunnel on MobileIron Core.
2. Use an existing certificate or generate a new one.
   If you have an existing certificate, see Use an existing certificate.
   Otherwise, see Generate a certificate.
3. Configure the Sentry with an AppTunnel service.
4. Configure the AppTunnel service in the AppConnect app configuration.

Enable AppTunnel on MobileIron Core

To enable AppTunnel on MobileIron Core:

1. In the Admin Portal, go to Settings.
2. Select Additional Products > Licensed Products.
3. Select AppConnect For Third-party And In-house Apps if it isn't already selected.
4. Select AppTunnel For Third-party And In-house Apps if it isn't already selected.
5. Click Save.

Use an existing certificate

Standalone Sentry only allows AppConnect apps on authenticated devices to use AppTunnel with HTTP/S tunneling. This device authentication involves:

- Providing Standalone Sentry with a root certificate.
- Providing the device with an identity certificate to present to the Standalone Sentry. The identity certificate is provisioned from the certificate authority (CA) that originated the root certificate.

If you already have an existing certificate, typically a .p12 file, you can use it for both purposes.

To upload the certificate to MobileIron Core:

1. In the Admin Portal, go to Policies & Configs > Configurations.
2. Select Add New > Certificate Enrollment > Single File Identity.
3. For Name, enter any name.
   For example: Tunneling Identiity Certificate
4. For Certificate 1, click Browse to select the .p12 or .pfx file of the identity certificate.
5. For Password 1, enter the password for the certificate's private key, if applicable.
6. Click Save.

Generate a certificate

Standalone Sentry only allows AppConnect apps on authenticated devices to use AppTunnel with HTTP/S tunneling. This device authentication involves:

- Providing Standalone Sentry with a root certificate.
- Providing the device with an identity cert to present to the Standalone Sentry. The identity cert is provisioned from the certificate authority (CA) that originated the root certificate.

One convenient way to get these certs iinvolves making MobileIron Core a local certificate authority (CA).

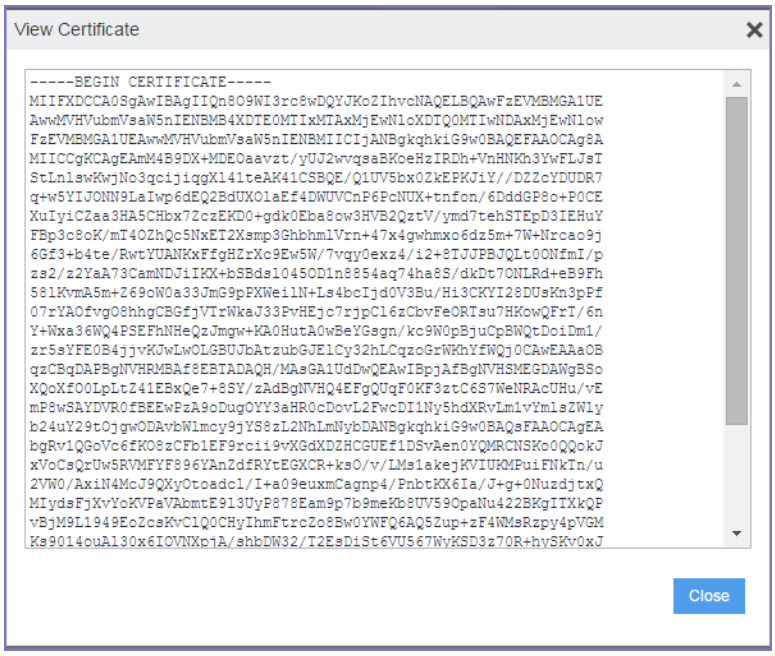
This process involves the following high-level steps:

1. Create a certificate authority for using AppTunnel with HTTP/S tunneling
2. Create a local certificate enrollment setting

Create a certificate authority for using AppTunnel with HTTP/S tunneling

To create a local certificate authority on MobileIron Core to be used in generating certificates:

1. In the Admin Portal, select Services > Local CA.
2. Select Add > Generate Self-Signed Cert
3. Enter a name for Local CA Name.
   For example: CA for AppTunnel
4. Set Key Length to 2048.
5. Set the Issuer Name to "CN=Tunneling CA".
6. Click Generate.
   A screen titled Certificate Template displays.
7. Click Save.
8. Click View Certificate next to your new local certificate authority.



9. Copy all the text in into a text file.
10. Save the text file.
    You will upload this text file later as the root certificate for authenticating devices to the Standalone Sentry.

Create a local certificate enrollment setting

After you configure MobileIron Core as a local CA, you create a local certificate enrollment setting. This setting configures MobileIron Core acting as a local CA to generate identity certificates for the devices to present to Standalone Sentry.

To create a local certificate enrollment setting:

1. In the Admin Portal, select Policies & Configs > Configurations
2. Select Add New > Certificate Enrollment > Local.
   A dialog appears entitled New Local Certificate Enrollment Setting.
3. Enter a descriptive name in the Name field.
   For example: Tunneling certificate
4. For Local CA, select the certificate authority you created for AppTunnel.
5. For Subject, enter “cn=tunneling”.
   The value can be any string.
6. For Key Length, select 2048.
7. Click Issue Test Certificate.
   The issued test certificate displays.
8. Click OK to close the displayed certificate.
9. Click Save to save the local certificate enrollment setting.

Configure the Sentry with an AppTunnel service

To support AppTunnel with HTTP/S tunneling, configure the Sentry with the internal servers that your app uses.

Do the following:

1. In the Admin Portal, go to Services > Sentry.
2. Select Add New > Standalone Sentry.
3. Enter the host name of the Sentry that MobileIron provides you.
4. Select Enable AppTunnel.
5. For Device Authentication Configuration:
   If you already had a certificate, select Group Certificate.
   If you created a local certificate authority, select Identity Certificate.
6. Click Upload Certificate.
   If you already had a certificate, upload it.
   If you created a local certificate authority, upload the certificate text file that you created in Create a certificate authority for using AppTunnel with HTTP/S tunneling. It is the root certificate for authenticating devices to the Standalone Sentry.
7. In the AppTunnel Configuration section, click + to add a new service.
8. Enter a Service Name.
   The service name is any unique identifier for the internal server or servers that your AppConnect app tunnels to. Entering <ANY> means that the app can reach any of your internal servers.
   Service Name examples:
   SharePoint
   HumanResources
9. For Server Auth, select Pass Through.
   This field selects the authentication scheme for the Standalone Sentry to use to authenticate the user to the internal server. Pass Through means that the Sentry passes through the authentication credentials, such as the user ID and password (basic authentication) or NTLM, to the internal server.

NOTE: The other option is Kerberos. Kerberos means that the Sentry uses Kerberos Constrained Delegation (KCD). The corporate environment must be set up for Kerberos Constrained Delegation.

10. Enter a Server List.
    Enter a semicolon-separated list of internal server host names or IP addresses and the port that the Sentry can access.
    For example:
    sharepoint1.companyname.com:443;sharepoint2.companyname.com:443.
    When you enter multiple servers, the Sentry uses a round-robin distribution to load balance the servers. That is, it sets up the first tunnel with the first internal server, the next with the next internal server, and so on.

NOTE: If you selected <ANY> for the Service Name, the Server List is not applicable.

11. Select TLS Enabled if the internal servers require SSL.
    Although port 443 is typically used for https and requires SSL, the internal server can use other port numbers requiring SSL.

NOTE: If you selected <ANY> for the Service Name, do not select TLS Enabled.

12. Do not fill in Server SPN List. It applies only when the Server Auth field is Kerberos.
13. Select Proxy/ATC only if your testing requires that you direct the AppTunnel service traffic through a proxy server. The proxy server is located behind the firewall and sits between the Sentry and corporate resources. This deployment allows you to access corporate resources without having to open the ports that Sentry would otherwise require.
    If selected, also configure the Server-side Proxy fields: Proxy Host Name / IP and Proxy Port.
14. Click Save.
15. Click View Certificate on the row with your new Sentry.

This action copies the Sentry's self-signed certificate that you created to MobileIron Core.

Configure the AppTunnel service in the AppConnect app configuration

The AppConnect app configuration specifies the AppTunnel services that your app uses. You configured these services on the Sentry.

To configure AppTunnel with HTTP/S tunneling on an AppConnect app configuration:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select Add New > AppConnect > App Configuration.

NOTE: If you already have created an AppConnect app configuration for your app, select it and click Edit in the right-hand pane.

3. Enter a name for the AppConnect app configuration if this is a new one.
   For example: My App's App Configuration
4. In the Application field, enter the bundle ID of your app if this is a new app configuration.
   For example: com.MyCompany.MySecureApp
5. In the AppTunnel Rules section, click Add+ to add a new AppTunnel configuration.
6. For Sentry, select the Sentry from the drop-down list.
7. For Service, select the service name from the drop-down list.
   You created this service name in Configure the Sentry with an AppTunnel service.
8. For the URL Wildcard, enter the host name or URL of the internal app server with which the app communicates. If the Service specified for this server in Configure the Sentry with an AppTunnel service is <ANY>, the host name can use the wildcard character *.
   If a URL request in your app matches the value you enter here, the request uses AppTunnel with HTTP/S tunneling.
   Examples:
   sharepoint1.yourcompany.com
   *.yourcompanyname.com
9. For Port, enter the port number that the app connects to.
10. For Identity Certificate:
    If you already had a certificate, select the certificate setting that you created in Use an existing certificate.
    If you created a local certificate authority, select the local certificate enrollment setting that you created in Create a local certificate enrollment setting. This selection will result in the device receiving an identity certificate from Core that it will present to the Standalone Sentry for device authentication.
11. Click Save.

If you are creating a new AppConnect app configuration:

1. Select the new AppConnect app configuration.
2. Select Actions > Apply To Label.
3. Select iOS.
4. Click Apply.
5. Click OK.

Push the changes to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.

4. Tap Force Device Check-in.

If your app is running, Mobile@Work launches and updates the AppConnect app configuration. If your app is not running, Mobile@Work launches and updates the configuration the next time that you run your app. When Mobile@Work has updated the configuration, your app will use AppTunnel with HTTP/S tunneling for the URLs you specified.

Verify that your app's networking capabilities work as expected.

Test logging messages to the console or files

Log levels

A MobileIron Core administrator can configure Core with the log level for your app. By default, the log level for an app is `ACLOGLEVEL_STATUS`.

The administrator has a choice of four log levels as shown in the following table:

TABLE 48. LOG LEVELS

Administrator log level for app	Corresponding ACLogLevel value for app
Status	ACLOGLEVEL_STATUS
Info	ACLOGLEVEL_INFO
Verbose	ACLOGLEVEL_VERBOSE
Debug	ACLOGLEVEL_DEBUG

Debug code for verbose and debug log levels

When the administrator chooses verbose or debug, the administrator also configures a debug code. The debug code is any string. Mobile@Work requires the device user to enter that string before changing the app's log level. This extra security is because messages logged at verbose and debug log levels may contain sensitive data.

Logging to files

The detailed log data for your AppConnect app, and the AppConnect library contained in the app, is logged to the device's console. The administrator can choose to write the log data for the app to files on the device in addition to writing the data to the device's console.

Log file details

Details regarding the log files for each app are:

- The log files for each app are saved to the following directory:
  Apps/<*app name*>/Library/Application Support/AppConnectLogs
- The log file for each app is named appConnect.log.
- The log file is at most 1 MB.
- When appconnect.log exceeds 1 MB:

1. It is renamed to appconnect.log.<*timestamp*>.
   Example: appconnect.log.2015-05-28 15:13:21
2. Logging begins in a new file named appconnect.log.
3. If 20 log files already exist, the oldest file is deleted.

Configuring logging to files

To log data to a file for an AppConnect app, add a key-value pair to the app's AppConnect app configuration:

1. In the Admin Portal, select **Policies & Configs > Configurations**
2. Select the app configuration for the app and click **Edit**.
   If the app does not already have an app configuration, select **Add New > AppConnect > App Configuration**.
   Enter a name and description for the new app configuration and the app's bundle ID.
3. In App-specific Configurations, click **Add+** to add a key-value pair.
4. Enter **MI_AC_ENABLE_LOGGING_TO_FILE** in the key field.
   The key name is case-sensitive.
5. Enter **Yes** in the value field.
6. Click **Add+** to add another key-value pair for the log level.
7. Enter **MI_AC_LOG_LEVEL** in the key field.
   The key name is case-sensitive.
8. Enter one of the following in the value field: error (the default), info, verbose, or debug.
   This value is not case-sensitive.
9. If you entered verbose or debug, click **Add+** to add another key-value pair.
10. Enter **MI_AC_LOG_LEVEL_CODE** in the key field.
    The key name is case-sensitive.
11. Enter a string for the value.
    The device user will enter this string to activate the verbose or debug log level. You can make up any string. For example, enter 37!8D. For the most security, use a code that is difficult to guess.
    The string is case-sensitive.
12. Click **Save**.

If you created a new AppConnect app configuration, apply the appropriate labels to it.

Pushing the new log level to the device

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap **Settings**.
3. Tap **Check for Updates**.
4. Tap **Force Device Check-in**.
   If your app is running, it receives the notification for the new configuration. Otherwise, it receives the notification the next time it runs. If the log level is verbose or debug, device user interaction is required to activate the new log level.

Verify that your app correctly handles the new log levels according to your app's requirements and design.

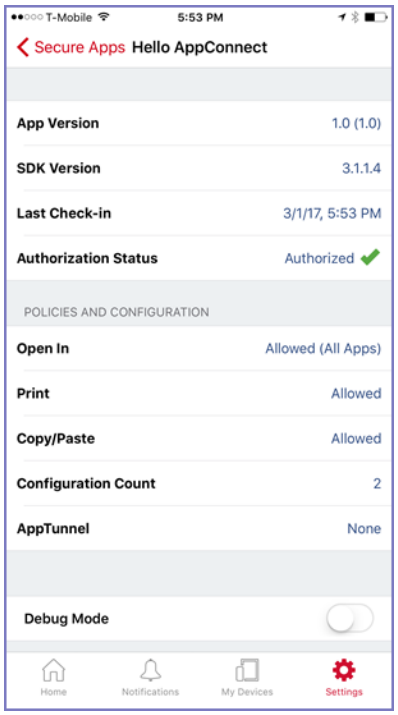
Activating verbose or debug logging on the device

Log levels verbose and debug require device user interaction. Your app is not notified of these log levels until the device user activates debug mode in Mobile@Work. This activation switch appears in Mobile@Work's detailed status display for your app. The detailed status display for your app is available after you have launched your app the first time.

The detailed status display for an AppConnect app includes a Debug Mode switch only when you have configured both of the following in the app's AppConnect app configuration:

- a log level of verbose or debug
- a debug code

In this case, a detailed status display screen for an AppConnect app shows the Debug Mode switch:



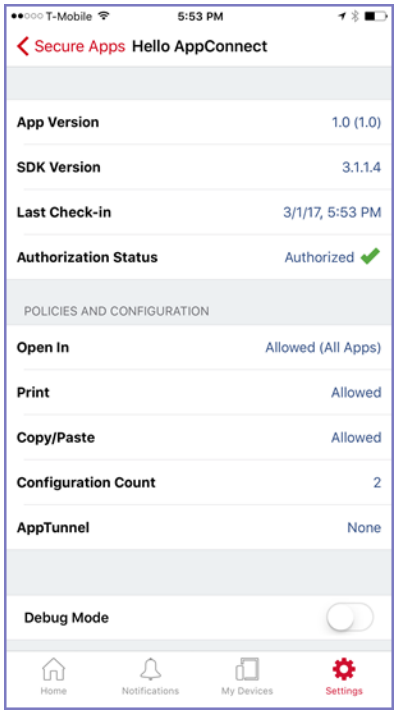
Screenshot from Mobile@Work 9.1

NOTE: Regarding the keys MI_AC_LOG_LEVEL and MI_AC_LOG_LEVEL_CODE:

- They are not included in the configuration count on an app's detailed status display.
- They are not included in the configuration your app receives through the AppConnect for iOS API.
- If the administrator makes changes to the AppConnect app configuration that involve only these keys, the AppConnect library does not call the `-appConnect:configChangedTo:` notification method.

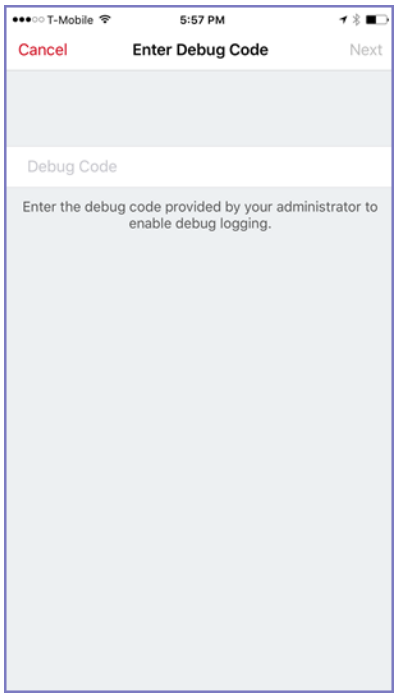
To activate verbose or debug level logging, do the following on the device:

1. Open Mobile@Work on the device.
2. Tap **Settings**.
3. Tap **Check For Updates**.
4. Tap **Force Device Check-In** to make sure that Mobile@Work has received the updated log level.
5. Tap **Settings**.
6. Tap **Secure Apps**.
7. Tap the app for which you want verbose or debug level logging.



Screenshot from Mobile@Work 9.1

8. Tap **Debug Mode**.



Screenshot from Mobile@Work 9.1

9. Enter the debug code.
10. Tap **Next**.

Verify that your app correctly handles the verbose and debug levels according to your app's requirements and design.

Verbose or debug level logging is activated for 24 hours, after which it is automatically deactivated the next time that you launch or switch to the app. However, you can deactivate it any time by tapping Debug Mode again. When deactivated, your app's log level returns to the default, which is `ACLOGLEVEL_STATUS`.

Sending log files in an email

You can use Mobile@Work for iOS to send log files to an email address of your choice as a convenient way to view the files. This feature requires Mobile@Work 9.8 for iOS through the most recently released version as supported by MobileIron.

Mobile@Work displays the option to send logs on the app's status details screen, available in Mobile@Work at **Settings > Secure Apps > *<app name>*.** The option is at the bottom of the screen with this text: **Send *<app name>* Logs.**

NOTE: The displayed option is disabled if the app's AppConnect authorization status is not authorized.

When the option is displayed and enabled, tapping it brings up the list of apps able to share the log files, such as email apps, only if all of the following are true:

- You included the key-value pairs for the app in its app configuration on MobileIron Core:
  - **MI_AC_LOG_LEVEL** set to **debug**

  - **MI_AC_LOG_LEVEL_CODE** set to a chosen string
  - **MI_AC_ENABLE_LOGGING_TO_FILE** set to **Yes**
- In Mobile@Work in **Settings > Secure Apps > <*app name*>,** you have turned on **Debug Mode** and entered the string from **MI_AC_LOG_LEVEL_CODE**.

Test the app documentation

A MobileIron Core administrator configures Core with information about your app. You provide this information in documentation about your app. Test whether your app correctly handles what your documentation specifies.

For more information, see Best Practices Using the AppConnect for iOS SDK.

Testing for In-house App Developers

In-house AppConnect app testing overview

Test your app using the instructions in this chapter or the instructions in Testing for Third-party App Developers based on the following table:

TABLE 49. WHERE TO FIND THE RIGHT TESTING INSTRUCTIONS

Your role	Testing instructions
In-house app developer whose organization uses MobileIron Core or Connected Cloud.	This chapter.
In-house app developer whose organization uses MobileIron Cloud	See Testing for Third-party App Developers
Third-party app developer	See Testing for Third-party App Developers

Testing with MobileIron Core as described in this chapter is necessary to verify the AppConnect-related functionality of your AppConnect app. If your app accesses servers behind a firewall using AppTunnel, a Standalone Sentry is necessary to verify the AppTunnel feature. All AppConnect apps require Mobile@Work to interact with Core.

As an in-house AppConnect app developer, contact your organization's Core administrator to get access to a Core and Standalone Sentry (if necessary) for testing. You then use a web portal called the Admin Portal to make configuration changes necessary for testing your app.

Mobile@Work is available from the Apple App Store.

Set up MobileIron Core

To set up MobileIron Core for testing your AppConnect app, do the following high-level steps:

1. Login to the Admin Portal.
2. Enable AppConnect on MobileIron Core.
3. Create a label for testing your app.
4. Upload your app to MobileIron Core if you use AppConnect.plist.
5. Verify your AppConnect.plist settings.
6. Configure the AppConnect global policy.
7. Create an AppConnect container policy, if necessary.

NOTE: These instructions are for Core 9.7.0.0.

Login to the Admin Portal

Contact your organization's MobileIron Core administrator to get the following information about the Core to test with:

- the URL for accessing the Core's Admin Portal
  The Admin Portal is a web portal for configuring Core. It has the format:
  https://<Core domain name>/mifs
- a username and password for accessing the Admin Portal
- a username and password for registering a device with Core
  Depending on your Core administrator, this username and password can be the same as the username and password for accessing the Admin Portal.

To login to Core:

1. Open a browser to the URL for accessing the Core's Admin Portal.
   For example:
   https://myCore.mycompany.com/mifs
2. Enter your Username and Password for accessing the Admin Portal.
3. Click Sign In.
   You are now in the Admin Portal.

Enable AppConnect on MobileIron Core

To test your AppConnect app, ensure that AppConnect is enabled on MobileIron Core.

1. In the Admin Portal, go to Settings.
2. Select Additional Products > Licensed Products.
3. Select AppConnect For Third-party And In-house Apps if it is not already selected.
4. Click Save.

Create a label for testing your app

MobileIron Core uses labels to associate policies and apps with devices. For testing your app, create a new label so that your testing impacts only your test device.

1. In the Admin Portal, go to Devices & Users > Labels.
2. Click Add Label.
3. Enter a name for the label.
   For example: AppConnect Test
4. Enter a description.
   For example: Use only for devices testing new AppConnect apps.
5. Select Manual for the Type.
6. Click Save.

Upload your app to MobileIron Core if you use AppConnect.plist

If your app uses an AppConnect.plist, upload your app to MobileIron Core. Uploading your app causes Core to create and populate an AppConnect container policy and AppConnect app configuration with the values you entered in the AppConnect.plist.

To upload your app:

1. In the Admin Portal, select Apps > App Catalog.
2. Select iOS for Platform.
3. Click Add+.
   The iOS Add App Wizard starts.
4. Click In-House.
5. Click Browse to select your app's .ipa file.
6. Click Next.
7. Click Next.
8. Click Finish.
   The app is now in Core's App Catalog. Core has created an AppConnect container policy and AppConnect app configuration based on your AppConnect.plist.
9. Select the row listing your app.
10. Select Actions > Apply To Label.
11. Select the label that your created in Create a label for testing your app.
12. Click Apply.
    Core applies the label to your app. It also applies it to the AppConnect container policy and AppConnect app configuration.

Verify your AppConnect.plist settings

Once you have uploaded your app to MobileIron Core, verify that the AppConnect.plist settings are correctly reflected in the AppConnect container policy and AppConnect app configuration.

To verify the AppConnect.plist settings:

1. On the Admin Portal, go to Policies & Configs > Configurations.
2. Select the row with the name of your app and the Setting Type APPCONFIG.

3. Click Edit in the right-hand pane.
4. In the App-specific Configurations section, verify the keys and values are what you entered in the AppConnect.plist.
5. Click Cancel.
6. Select the row with the name of your app and the Setting Type APPPOLICY.
7. Click Edit in the right-hand pane.
8. Verify the data loss prevention settings are what you entered in the AppConnect.plist.
9. Click Cancel.

If any of the key-value pairs or data loss prevention policies are not what you expected, review the contents of your AppConnect.plist.

Configure the AppConnect global policy

An AppConnect global policy is necessary for your AppConnect app to work properly.

To configure an AppConnect global policy:

1. In the Admin Portal, select Policies & Configs > Policies.
2. Select Add New > AppConnect.
3. Enter a name for the AppConnect global policy.
   For example: Test AppConnect Global Policy.
4. For AppConnect, select Enabled.
   The display now shows all the AppConnect global policy fields.
5. In the AppConnect Passcode section, for Passcode Type, select Numeric.
6. In the AppConnect Passcode section, select Passcode Is Required For iOS Devices.
7. Click Save.
   The dialog box closes and the new AppConnect global policy appears in the list.
8. Select the AppConnect global policy that you just created.
9. Select More Actions > Apply To Label.
10. Select the test label that you created in Create a label for testing your app.
11. Click Apply.
12. Click OK.

NOTE: Do not select Authorize in the field Apps Without An AppConnect Container Policy in the section Data Loss Prevention Policies in the AppConnect global policy. You will authorize the app with an AppConnect container policy instead.

Create an AppConnect container policy, if necessary

An app is authorized only if an AppConnect container policy for the app is present on the device. If you have an AppConnect.plist in your app, and uploaded the app to MobileIron Core, Core creates an AppConnect container policy automatically. If you do not have an AppConnect.plist in your app, manually create an AppConnect container policy.

To create an AppConnect container policy:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select Add New > AppConnect > Container Policy.

3. Enter a name for the AppConnect container policy.
   For example: My App's Container Policy
4. In the Application field, enter the bundle ID of your app.
   For example: com.MyCompany.MySecureApp
5. Click Save.
   The dialog box closes and the new AppConnect container policy appears in the list.
6. Select the AppConnect container policy you just created.
7. Select Actions > Apply To Label.
8. Select the test label that you created in Create a label for testing your app.
9. Click Apply.
10. Click OK.

Set up your end-user device

To set up your end-user device, do the following high-level steps:

1. Set up Mobile@Work on an iOS device.
2. Install your app on the device.
3. Set up the AppConnect passcode on the device.

Set up Mobile@Work on an iOS device

To set up Mobile@Work for iOS on your device:

1. Download and install Mobile@Work from the Apple App Store.
2. Tap the MobileIron app icon to launch Mobile@Work.
3. Enter the user name that the Core administrator gave you for registering your test device.
4. Enter the server name that the Core administrator gave you.
   For example: myCore.mycompany.com
5. Enter the password.
   Enter the password that the Core administrator gave you for registering your test device.
6. Follow the prompts from Mobile@Work to complete its setup.
   Allow Mobile@Work to use the current location.
   Install new profiles and certificates when prompted.

Install your app on the device

Install your app on the device in the same way you install any app that you are testing.

Set up the AppConnect passcode on the device

When you run your app for the first time, Mobile@Work prompts you to create the AppConnect passcode. Follow the steps to create the AppConnect passcode.

Test authorization status handling

You can make changes to the MobileIron Core configuration to test your app's handling of the different authorization statuses: authorized, unauthorized, and retired.

Change the status to authorized or unauthorized

A security policy on MobileIron Core specifies the requirements for a device. If a device is not compliant with a requirement, the security policy specifies a compliance action. One compliance action is to block AppConnect apps on the device, which means that the apps become unauthorized.

The list of requirements that can impact authorization is long, but for testing your app, you need to work with only one requirement. The requirement involves a list of device models that are not allowed to use AppConnect apps.

Therefore, to unauthorize the app on the device:

1. In the Admin Portal, select Policies & Configs > Policies.
2. Select Add New > Security.
3. Enter a name.
   For example: AppConnect test security policy
4. Scroll down to the section called Access Control, under For iOS Devices.
5. Select Block Email, AppConnect Apps, And Send Alert For The Following Disallowed Devices.
6. Move the model of your test device to the Disallowed area.
7. Click Save.
   Core creates the new security policy.
8. Select the row listing the new security policy.
9. Select More Actions > Apply To Label.
10. Select the test label that you created in Create a label for testing your app.
11. Click Apply.
12. Click OK.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is unauthorized. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the change to the unauthorized state. Specifically, verify that your app:

- exits any sensitive part of the application.
- stops allowing the user to access sensitive data and views.
- displays the message received in the callback method that explains the authorization status change.
- calls the `-authStateApplied:message:` method.

To re-authorize the app on the device:

1. In the Admin Portal, select Policies & Configs > Policies.

2. Select the security policy that you created.
3. Click Edit in the right-hand pane.
4. In the section called Access Control, under For iOS Devices, *uncheck* Block Email, AppConnect Apps, And Send Alert For The Following Disallowed Devices.
5. Click Save.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is authorized. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the change to the authorized state. Specifically, verify that your app:

- allows the user to access sensitive data and views.
- calls the `-authStateApplied:message:` method.

Change the status to retired

An app is authorized only if an AppConnect container policy for the app is present on the device. If you remove the AppConnect container policy from the device, the app becomes retired.

To retire the app on the device:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select the AppConnect container policy for your app.
3. Select Actions > Remove From Label.
4. Select the label that you created in Create a label for testing your app.
5. Click Remove.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is retired. Otherwise, it receives the notification the next time it runs. The message string in the notification is the default unauthorized message:
   “Your administrator has not authorized this app.”

Verify that your app correctly handles the change to the retired state. Specifically, verify that your app:

- exits any sensitive part of the application.
- deletes all sensitive data, including any stored authentication credentials, data in files, keychain items, pasteboard data, and any other persistent storage.
- displays the message received in the callback method that explains the authorization status change.
- calls the `-authStateApplied:message:` method.

Reauthorize a retired app

A retired app is sometimes re-authorized at a later time.

To reauthorize the retired app on the device:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select the AppConnect container policy for your app.
3. Select Actions > Apply To Label.
4. Select the label that you created in Create a label for testing your app.
5. Click Apply.
6. Click OK.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification that it is authorized. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the change to the authorized state. Specifically, verify that your app:

- dismisses any user interface that displays that the user is not authorized to use the app.
- allows the user to access sensitive data and views.
- calls the `-authStateApplied:message:` method.

Test data loss prevention policy handling

The AppConnect container policy for your app specifies its data loss prevention (DLP) policies. In this policy, you specify whether your app is allowed to:

- copy content to the iOS pasteboard.
- drag and drop content to other apps
- print by using AirPrint, any future iOS printing feature, any current or future third-party libraries or apps that provide printing capabilities.
- share documents with other apps.

By changing the AppConnect container policy, you can test:

- your app's behavior for each data loss prevention policy.
- how your app handles changes to the policies in the notification callback methods in the AppDelegateProtocol.

To change the DLP policies:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select the AppConnect container policy for your app.
3. Click Edit in the right-hand pane.
4. Allow or prohibit features relating to data loss prevention policies as follows:

TABLE 50. DLP POLICY DESCRIPTIONS

DLP policy	Description
Allow Print	Select Allow Print if you want the app to use the device's print capabilities.
Allow Copy/Paste to	Select Allow Copy/Paste to if you want the device user to be able to copy content from the AppConnect app to other apps. When you select this option, then select either: • All Apps Select All Apps if you want the device user to be able to copy content from the AppConnect app and paste it into any other app. • AppConnect Apps Select AppConnect Apps if you want the device user to be able to copy content from the AppConnect app and paste it into only other AppConnect apps.
Allow Drag and Drop	Select Allow Drag and Drop if you want the device user to be able to drag content from the AppConnect app to other apps. When you select this option, then select either: • All Apps Select All Apps if you want the device user to be able to drag content from the AppConnect app to any other app. • AppConnect Apps Select AppConnect Apps if you want the device user to be able to drag content from the AppConnect app to only other AppConnect apps.
Allow Open In	Select Allow Open In if you want the app to be allowed to use the device's Open In (document interaction) feature. When you select this option, then select either: • All Apps Select All Apps if you want the app to be able to send documents to any other app. • AppConnect Apps Select AppConnect Apps to allow an AppConnect app to send documents to only other AppConnect apps. NOTE: This option results in the `openInPolicy` property having the value `ACOPENINPOICY_WHITELIST`. Also, the `openInWhitelist` property contains the list of currently authorized AppConnect apps. • Whitelist Select Whitelist if you want the app to be able to send documents only to the apps that you specify. Enter the bundle ID of each app, one per line, or in a semicolon delimited list. For example: com.myAppCo.myApp1 com.myAppCo.myApp2;com.myAppCo.myApp3 The bundle IDs that you enter are case sensitive.

5. Click Save.
6. Click Yes to confirm.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notifications for the updated DLP policies. Otherwise, it receives the notifications the next time it runs.

Verify that your app correctly handles the data loss prevention policy changes, as shown in the following table:

TABLE 51. WHAT TO VERIFY WHEN A DLP POLICY CHANGES

Policy change	What to verify
Allow copy/paste to	• Verify that the user can cut or copy text, images, or other data to the pasteboard. • Where appropriate, verify that any special user interface that offers the ability to cut or copy data is available and enabled. Also, verify that your app calls the `-pasteboardPolicyApplied:message:` method.
Allow copy/paste to for AppConnect Apps only	• Verify that the user can cut or copy text, images, or other data to the pasteboard. • Where appropriate, verify that any special user interface that offers the ability to cut or copy data is available and enabled. • Verify that the user can paste the data from the pasteboard only into other AppConnect apps. Also, verify that your app calls the `-pasteboardPolicyApplied:message:` method.
Do not allow copy/paste to	• Verify that the user cannot to cut or copy text, images, or other data to the pasteboard. • Where appropriate, verify that any special user interface that offers the ability to cut or copy data is removed or disabled. • Verify your implementation of the callback method `-appConnect:copyAttemptedWhenUnauthorized:`. Also, verify that your app calls the `-pasteboardPolicyApplied:message:` method.
Allow drag and drop to only AppConnect apps	Verify your implementation of the callback method - `appConnectAttemptedDragAndDropToUnauthorizedApp:`.
Allow open in for all apps	Verify that your app enables user interfaces, if any, that give the user the option to use Open In. Also, verify that your app calls the `-openInPolicyApplied:message:` method.
Allow open in for AppConnect	Verify that:

TABLE 51. WHAT TO VERIFY WHEN A DLP POLICY CHANGES (CONT.)

Policy change	What to verify
apps	• your app enables user interfaces, if any, that give the user the option to use Open In. • your app calls the `-openInPolicyApplied:message:` method. • the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method, if implemented, behaves as you expect.
Allow open in for whitelisted apps	Verify that: • your app enables user interfaces, if any, that give the user the option to use Open In. • your app calls the `-openInPolicyApplied:message:` method. • the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method, if implemented, behaves as you expect.
Do not allow open in	Verify that: • your app disables user interfaces, if any, that give the user the option to use Open In. • your app calls the `-openInPolicyApplied:message:` method. • the `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` callback method, if implemented, behaves as you expect.
Allow print	For each part of your app that allows the user to print secure data, verify the capability is enabled. Also, verify that your app calls the `-printPolicyApplied:message:` method.
Do not allow print	For each part of your app that allows the user to print secure data, verify the capability is removed or disabled. Also, verify that your app calls the `-printPolicyApplied:message:` method.

Test AppConnect configuration change handling

AppConnect app configuration on MobileIron Core specifies key-value pairs for configuring your app. You add, and edit, key-value pairs using the Admin Portal.

By changing the AppConnect app configuration, you can test your app's `-appConnect:configChangedTo:` method in the AppDelegateProtocol.

If your app includes an AppConnect.plist, and you uploaded your app to Core, Core already has created a default AppConnect app configuration. Go to Update the AppConnect app configuration.

If your app does not include an AppConnect.plist, create an AppConnect app configuration.

Create an AppConnect app configuration

To create an AppConnect app configuration:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select Add New > AppConnect > App Configuration.
3. Enter a name for the AppConnect app configuration.
   For example: My App's App Configuration
4. In the Application field, enter the bundle ID of your app.
   For example: com.MyCompany.MySecureApp
5. In the App-specific Configurations section, click Add**+** to add a key-value pair.
6. Enter the key-value pairs:

TABLE 52. KEY-VALUE PAIRS IN AN APPCONNECT APP CONFIGURATION

Key	The key is any string that the app recognizes as a configurable item. For example: userid, appURL
Value	Enter the value. The value is either: • a string The string can have any value that is meaningful to the app. It can also include one or more of these MobileIron Core variables: $USERID$, $EMAIL$, $USER_CUSTOM1$, $USER_CUSTOM2$, $USER_CUSTOM3$, $USER_CUSTOM4$. If you do not want to provide a value, enter $NULL$. The $NULL$ value tells the app that the app user will need to provide the value. Examples: $USERID$ https://someEnterpriseURL.com • a Certificate Enrollment or Certificates setting Certificate Enrollment and Certificate settings that are configured in Policies & Configs > Configurations appear in the dropdown list. When you choose a Certificate Enrollment or Certificate setting, Core sends the contents of the certificate as the value. The contents are base64-encoded. If the certificate is password-encoded, Core automatically sends another key-value pair. The key's name is the string <*name of key for certificate*>_MI_CERT_PW. The value is the certificate's password.

7. Click Save.
8. Click Yes to confirm.
9. Select the new AppConnect app configuration.
10. Select Actions > Apply To Label.
11. Select the label that you created in Create a label for testing your app.
12. Click Apply.
13. Click OK.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.

3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification for the new configuration. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the new configuration, correctly applying and using the configured options according to your app's requirements and design.

Update the AppConnect app configuration

To update the AppConnect app configuration:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select the your app's AppConnect app configuration.
3. Click Edit in the right-hand pane.
4. In the App-specific Configurations section, click Add**+** to add a key-value pair. To delete a key-value pair, click the X on the row.
5. Update the key-value pairs as described in Create an AppConnect app configuration.
6. Click Save.
7. Click Yes to confirm.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.
   If your app is running, it receives the notification for the updated configuration. Otherwise, it receives the notification the next time it runs.

Verify that your app correctly handles the updated configuration, correctly applying and using the configured options according to your app's requirements and design.

Test using AppTunnel

Using MobileIron's AppTunnel feature, your app can securely tunnel HTTP and HTTPS network connections from the app to servers behind an organization's firewall. Your app does not take any special actions related to tunneling; the AppConnect library, Mobile@Work, and a Standalone Sentry handle tunneling for the app.

You can test the HTTP/S tunneling capability using the provided MobileIron Core and Sentry. Using the Admin Portal, you configure app-specific AppTunnel settings for Core and Sentry.

Before you begin: Contact your Core administrator to find out the host name or IP address of the Sentry to use for the AppTunnel feature.

To test your app's use of AppTunnel with HTTP/S tunneling, do these high-level steps:

1. Enable AppTunnel on MobileIron Core.
2. Use an existing certificate or generate a new one.

If you have an existing certificate, see Use an existing certificate.
Otherwise, see Generate a certificate.

3. Configure the Sentry with an AppTunnel service.
4. Configure the AppTunnel service in the AppConnect app configuration.

Enable AppTunnel on MobileIron Core

To enable AppTunnel on MobileIron Core if it isn't already enabled:

1. In the Admin Portal, go to Settings.
2. Select Additional Products > Licensed Products.
3. Select AppConnect For Third-party And In-house Apps if it isn't already selected.
4. Select AppTunnel For Third-party And In-house Apps if it isn't already selected.
5. Click Save.

Use an existing certificate

Standalone Sentry only allows AppConnect apps on authenticated devices to use AppTunnel with HTTP/S tunneling. This device authentication involves:

- Providing Standalone Sentry with a root certificate.
- Providing the device with an identity cert to present to the Standalone Sentry. The identity cert is provisioned from the certificate authority (CA) that originated the root certificate.

To upload the certificate to MobileIron Core:

1. In the Admin Portal, go to Policies & Configs > Configurations.
2. Select Add New > Certificate Enrollment > Single File Identity.
3. For Name, enter any name.
   For example: Tunneling Identiity Certificate
4. For Certificate 1, click Browse to select the .p12 or .pfx file of the identity certificate.
5. For Password 1, enter the password for the certificate's private key, if applicable.
6. Click Save.

Generate a certificate

Standalone Sentry only allows AppConnect apps on authenticated devices to use AppTunnel with HTTP/S tunneling. This device authentication involves:

- Providing Standalone Sentry with a root certificate.
- Providing the device with an identity cert to present to the Standalone Sentry. The identity cert is provisioned from the certificate authority (CA) that originated the root certificate.

One convenient way to get these certs involves making MobileIron Core a local certificate authority (CA).

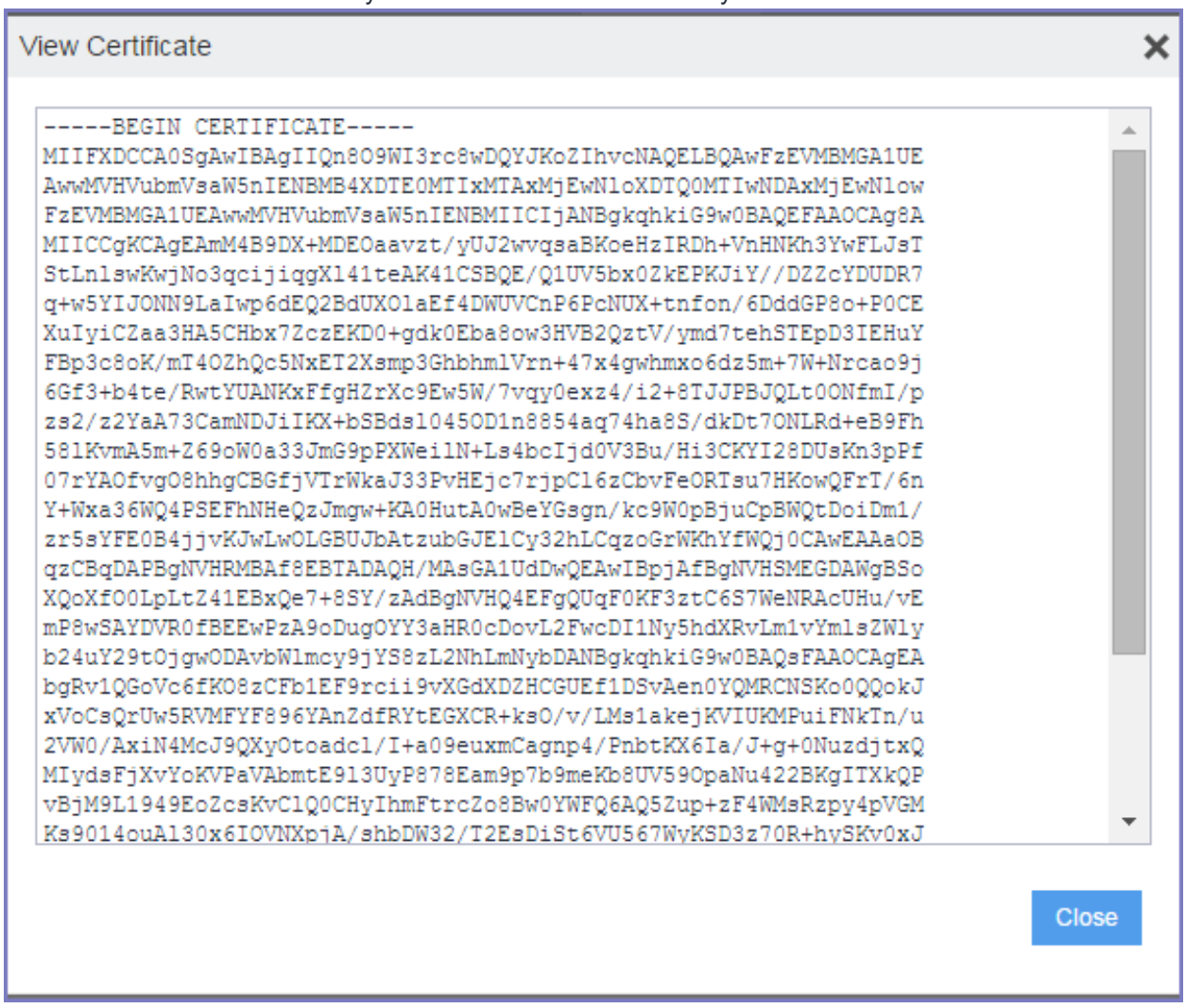
This process involves the following high-level steps:

1. Create a certificate authority for using an AppTunnel with HTTP/S tunneling
2. Create a local certificate enrollment setting

Create a certificate authority for using an AppTunnel with HTTP/S tunneling

To create a local certificate authority on MobileIron Core to be used in generating certificates:

1. In the Admin Portal, select Services > Local CA.
2. Select Add > Generate Self-Signed Cert
3. Enter a name for Local CA Name.
   For example: CA for AppTunnel
4. Set Key Length to 2048.
5. Set the Issuer Name to “CN=Tunneling CA”.
6. Click Generate.
   A screen titled Certificate Template displays.
7. Click Save.
8. Click View Certificate next to your new local certificate authority.



9. Copy all the text into a text file.
10. Save the text file.
    You will upload this text file later as the root certificate for authenticating devices to the Standalone Sentry.

Create a local certificate enrollment setting

After you configure MobileIron Core as a local CA, you create a local certificate enrollment setting. This setting configures MobileIron Core acting as a local CA to generate identity certificates for the devices to present to Standalone Sentry.

To create a local certificate enrollment setting:

1. In the Admin Portal, select Policies & Configs > Configurations
2. Select Add New > Certificate Enrollment > Local.
   A dialog appears entitled New Local Certificate Enrollment Setting.
3. Enter a descriptive name in the Name field.
   For example: Tunneling certificate
4. For Local CA, select the certificate authority you created for AppTunnel.
5. For Subject, enter “cn=tunneling”.
   The value can be any string.
6. For Key Length, select 2048.
7. Click Issue Test Certificate.
   The issued test certificate displays.
8. Click OK to close the displayed certificate.
9. Click Save to save the local certificate enrollment setting.

Configure the Sentry with an AppTunnel service

To support AppTunnel with HTTP/S tunneling, configure the Sentry with the internal servers that your app uses.

Do the following:

1. In the Admin Portal, go to Services > Sentry.
2. Click the edit icon next to the Sentry that your MobileIron Core Administrator has designated for your AppTunnel testing.
3. Select Enable AppTunnel if it is not already selected.
4. For Device Authentication Configuration:
   If you already had a certificate, select Group Certificate.
   If you created a local certificate authority, select Identity Certificate.
5. Click Upload Certificate.
   If you already had a certificate, upload it.
   If you created a local certificate authority, upload the certificate text file that you created in Create a certificate authority for using an AppTunnel with HTTP/S tunneling. It is the root certificate for authenticating devices to the Standalone Sentry.
6. In the AppTunnel Configuration section, click + to add a new service.
7. Enter a Service Name.
   The service name is any unique identifier for the internal server or servers that your AppConnect app tunnels to.
   Entering <ANY> means that the app can reach any of your internal servers.
   Service Name examples:
   SharePoint
   HumanResources
8. For Server Auth, select Pass Through.

This field selects the authentication scheme for the Standalone Sentry to use to authenticate the user to the internal server. Pass Through means that the Sentry passes through the authentication credentials, such as the user ID and password (basic authentication) or NTLM, to the internal server.

NOTE: The other option is Kerberos. Kerberos means that the Sentry uses Kerberos Constrained Delegation (KCD). The corporate environment must be set up for Kerberos Constrained Delegation.

9. Enter a Server List.
   Enter a semicolon-separated list of internal server host names or IP addresses and the port that the Sentry can access.
   For example:
   sharepoint1.companyname.com:443;sharepoint2.companyname.com:443.
   When you enter multiple servers, the Sentry uses a round-robin distribution to load balance the servers. That is, it sets up the first tunnel with the first internal server, the next with the next internal server, and so on.

NOTE: If you selected <ANY> for the Service Name, the Server List is not applicable.

10. Select TLS Enabled if the internal servers require SSL.
    Although port 443 is typically used for https and requires SSL, the internal server can use other port numbers requiring SSL.

NOTE: If you selected <ANY> for the Service Name, do not select TLS Enabled.

11. Do not fill in Server SPN List. It applies only when the Server Auth field is Kerberos.
12. Select Proxy/ATC only if your testing requires that you direct the AppTunnel service traffic through a proxy server. The proxy server is located behind the firewall and sits between the Sentry and corporate resources. This deployment allows you to access corporate resources without having to open the ports that Sentry would otherwise require.
    If selected, also configure the Server-side Proxy fields: Proxy Host Name / IP and Proxy Port.
13. Click Save.
14. Click View Certificate on the row with your new Sentry.
    This action copies the Sentry's self-signed certificate that you created to Core.

Configure the AppTunnel service in the AppConnect app configuration

The AppConnect app configuration specifies the AppTunnel services that your app uses. You configured these services on the Sentry.

To configure AppTunnel on an AppConnect app configuration:

1. In the Admin Portal, select Policies & Configs > Configurations.
2. Select Add New > AppConnect > App Configuration.

NOTE: If you already have an AppConnect app configuration for your app, select it and click Edit in the right-hand pane.

3. Enter a name for the AppConnect app configuration if this is a new one.
   For example: My App's App Configuration
4. In the Application field, enter the bundle ID of your app if this is a new app configuration.
   For example: com.MyCompany.MySecureApp
5. In the AppTunnel Rules section, click Add+ to add a new AppTunnel configuration.
6. For Sentry, select the Sentry from the drop-down list.
7. For Service, select the service name from the drop-down list.
   You created this service name in Create a certificate authority for using an AppTunnel with HTTP/S tunneling.

8. For the URL Wildcard, enter the host name or URL of the app server with which the app communicates. If the Service specified for this server in Configure the Sentry with an AppTunnel service is <ANY>, the host name can use the wildcard character *.
   If a URL request in your app matches the value you enter here, the request uses AppTunnel with HTTP/S tunneling.
   Examples:
   sharepoint1.yourcompany.com
   *.yourcompanyname.com
9. For Port, enter the port number that the app connects to.
10. For Identity Certificate:
    If you already had a certificate, select the certificate setting that you created in Use an existing certificate.
    If you created a local certificate authority, select the local certificate enrollment setting that you created in Create a local certificate enrollment setting. This selection will result in the device receiving an identity certificate from Core that it will present to the Standalone Sentry for device authentication.
11. Click Save.

If you are creating a new AppConnect app configuration:

1. Select the new AppConnect app configuration.
2. Select Actions > Apply To Label.
3. Select the label that you created in Create a label for testing your app.
4. Click Apply.
5. Click OK.

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap Settings.
3. Tap Check for Updates.
4. Tap Force Device Check-in.

If app is running, Mobile@Work launches and updates the AppConnect app configuration. If your app is not running, Mobile@Work launches and updates the configuration the next time that you run your app. When Mobile@Work has updated the configuration, your app will use AppTunnel with HTTP/S tunneling for the URLs you specified.

Verify that your app's networking capabilities work as expected.

Test logging messages to the console or files

- Log levels
- Debug code for verbose and debug log levels
- Logging to files
- Log file details
- Configuring logging to files
- Pushing the new log level to the device
- Activating verbose or debug logging on the device
- Sending log files in an email

Log levels

A MobileIron Core administrator can configure Core with the log level for your app. By default, the log level for an app is `ACLOGLEVEL_STATUS`.

The administrator has a choice of four log levels as shown in the following table:
:

TABLE 53. LOG LEVELS

Administrator log level for app	Corresponding ACLogLevel value for app
Status	ACLOGLEVEL_STATUS
Info	ACLOGLEVEL_INFO
Verbose	ACLOGLEVEL_VERBOSE
Debug	ACLOGLEVEL_DEBUG

Debug code for verbose and debug log levels

When the administrator chooses verbose or debug, the administrator also configures a debug code. The debug code is any string. Mobile@Work requires the device user to enter that string before changing the app's log level. This extra security is because messages logged at verbose and debug log levels may contain sensitive data.

Logging to files

The detailed log data for your AppConnect app, and the AppConnect library contained in the app, is logged to the device's console. The administrator can choose to write the log data for the app to files on the device in addition to writing the data to the device's console.

Log file details

Details regarding the log files for each app are:

1. The log files for each app are saved to the following directory:
   Apps/<*app name*>/Library/Application Support/AppConnectLogs
- The log file for each app is named appConnect.log.
- The log file is at most 1 MB.
- When appconnect.log exceeds 1 MB:

1. It is renamed to appconnect.log.<*timestamp*>.
   Example: appconnect.log.2015-05-28 15:13:21
2. Logging begins in a new file named appconnect.log.
3. If 20 log files already exist, the oldest file is deleted.

Configuring logging to files

To log data to a file for an AppConnect app, add a key-value pair to the app's AppConnect app configuration:

1. In the Admin Portal, select **Policies & Configs > Configurations**
2. Select the app configuration for the app and click **Edit**.
   If the app does not already have an app configuration, select **Add New > AppConnect > App Configuration**. Enter a name and description for the new app configuration and the app's bundle ID.
3. In App-specific Configurations, click **Add+** to add a key-value pair.
4. Enter **MI_AC_ENABLE_LOGGING_TO_FILE** in the key field.
   The key name is case-sensitive.
5. Enter **Yes** in the value field.
6. Click **Add+** to add another key-value pair for the log level.
7. Enter **MI_AC_LOG_LEVEL** in the key field.
   The key name is case-sensitive.
8. Enter one of the following in the value field: error (the default), info, verbose, or debug.
   This value is not case-sensitive.
9. If you entered verbose or debug, click **Add+** to add another key-value pair.
10. Enter **MI_AC_LOG_LEVEL_CODE** in the key field.
    The key name is case-sensitive.
11. Enter a string for the value.
    The device user will enter this string to activate the verbose or debug log level. You can make up any string. For example, enter 37!8D. For the most security, use a code that is difficult to guess.
    The string is case-sensitive.
12. Click **Save**.

If you created a new AppConnect app configuration, apply the appropriate labels to it.

Pushing the new log level to the device

Push the change to your device immediately, by doing the following steps on the device:

1. Launch Mobile@Work.
2. Tap **Settings**.
3. Tap **Check for Updates**.
4. Tap **Force Device Check-in**.
   If your app is running, it receives the notification for the new configuration. Otherwise, it receives the notification the next time it runs. If the log level is verbose or debug, device user interaction is required to activate the new log level.

Verify that your app correctly handles the new log levels according to your app's requirements and design.

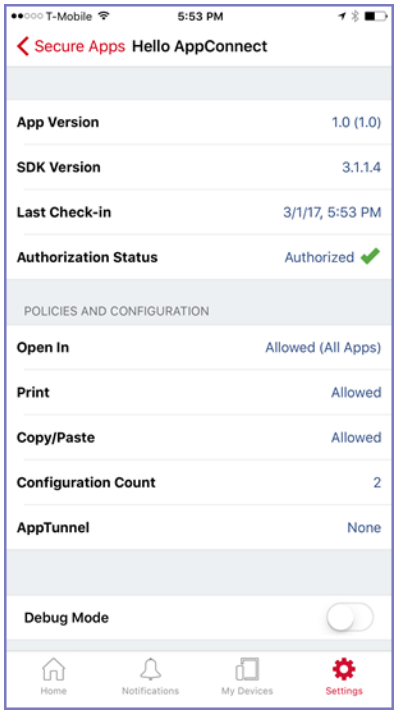
Activating verbose or debug logging on the device

Log levels verbose and debug require device user interaction. Your app is not notified of these log levels until the device user activates debug mode in Mobile@Work. This activation switch appears in Mobile@Work's detailed status display for your app. The detailed status display for your app is available after you have launched your app the first time.

The detailed status display for an AppConnect app includes a Debug Mode switch only when you have configured both of the following in the app's AppConnect app configuration:

- a log level of verbose or debug
- a debug code

In this case, a detailed status display screen for an AppConnect app shows the Debug Mode switch:



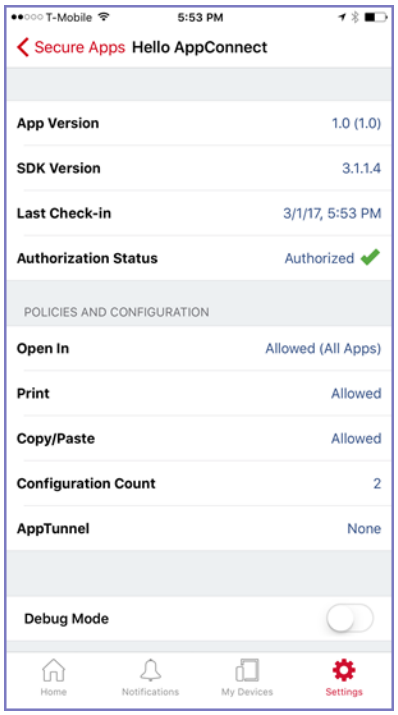
Screenshot from Mobile@Work 9.1

NOTE: Regarding the keys MI_AC_LOG_LEVEL and MI_AC_LOG_LEVEL_CODE:

- They are not included in the configuration count on an app's detailed status display.
- They are not included in the configuration your app receives through the AppConnect for iOS API.
- If the administrator makes changes to the AppConnect app configuration that involve only these keys, the AppConnect library does not call the `-appConnect:configChangedTo:` notification method.

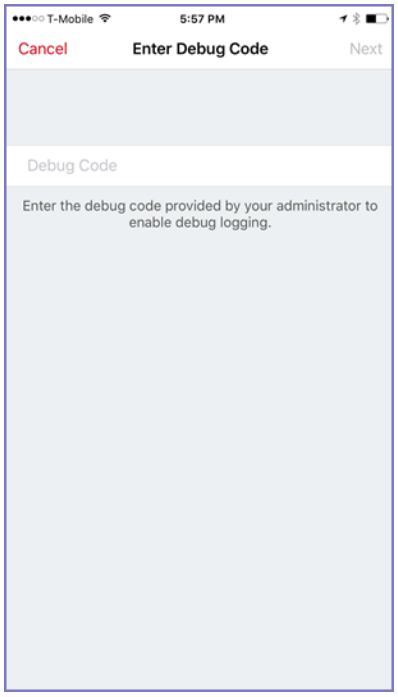
To activate verbose or debug level logging, do the following on the device:

1. Open Mobile@Work on the device.
2. Tap **Settings**.
3. Tap **Check For Updates**.
4. Tap **Force Device Check-In** to make sure that Mobile@Work has received the updated log level.
5. Tap **Settings**.
6. Tap **Secure Apps**.
7. Tap the app for which you want verbose or debug level logging.



Screenshot from Mobile@Work 9.1

8. Tap **Debug Mode**.



Screenshot from Mobile@Work 9.1

9. Enter the debug code.
10. Tap **Next**.

Verify that your app correctly handles the verbose and debug levels according to your app's requirements and design.

Verbose or debug level logging is activated for 24 hours, after which it is automatically deactivated the next time that you launch or switch to the app. However, you can deactivate it any time by tapping Debug Mode again. When deactivated, your app's log level returns to the default, which is `ACLOGLEVEL_STATUS`.

Sending log files in an email

You can use Mobile@Work for iOS to send log files to an email address of your choice as a convenient way to view the files.. This feature requires Mobile@Work 9.8 for iOS through the most recently released version as supported by MobileIron.

Mobile@Work displays the option to send logs on the app's status details screen, available in Mobile@Work at **Settings > Secure Apps > *<app name>*.** The option is at the bottom of the screen with this text: **Send *<app name>* Logs.**

NOTE: The displayed option is disabled if the app's AppConnect authorization status is not authorized.

When the option is displayed and enabled, tapping it brings up the list of apps able to share the log files, such as email apps, only if all of the following are true:

- You included the key-value pairs for the app in its app configuration on MobileIron Core:
  - **MI_AC_LOG_LEVEL** set to **debug**
  - **MI_AC_LOG_LEVEL_CODE** set to a chosen string
  - **MI_AC_ENABLE_LOGGING_TO_FILE** set to **Yes**
- In Mobile@Work in **Settings > Secure Apps > *<app name>*,** you have turned on **Debug Mode** and entered the string from **MI_AC_LOG_LEVEL_CODE**.

Test the app documentation

Once your app is ready for in-house distribution, a MobileIron Core administrator configures Core with information about your app. You provide this information in documentation about your app. Test whether your app correctly handles what your documentation specifies.

For more information, see Best Practices Using the AppConnect for iOS SDK.

Derived Credential Handling

Derived credential handling overview

Only use the APIs relating to derived credentials if you are developing an app that obtains derived credentials from a derived credential provider and delivers the credentials to the MobileIron client.

A derived credential is derived from the primary credential on a user's smart card and stored on the user's mobile device. The derived credential contains X.509 public key identity certificates derived from the primary credential's identity certificates.

The APIs allow your app to:

- Send a derived credential to the MobileIron client.
- Receive a request from the MobileIron client to get a new derived credential and deliver it to the MobileIron client.

Besides implementing this derived credential capability, your app must implement the necessary AppConnect APIs to behave as an AppConnect app.

Regarding derived credentials, when your app decides to get a derived credential, such as due to user interaction, your app does the following high-level steps:

1. Makes sure that the MobileIron client is installed and that it supports derived credentials.
2. Makes sure that sending derived credentials to the MobileIron client is currently allowed.
3. Obtains a derived credential from the derived credential provider.
4. Indicate which certificate in the derived credential is for what kind of use by AppConnect apps. The uses are authentication, signing, and encryption.
5. Sends the derived credential to the MobileIron client.

After the MobileIron client has the derived credential, AppConnect apps on the device can use the certificates that comprise the derived credential. Whether the AppConnect apps use the derived credential's certificates or other certificates depends on configuration settings on the MobileIron server.

Also, at any time, the MobileIron client can request a new derived credential from your app. At that time, your app repeats the above steps.

Derived credential header files

The following header files in the AppConnect.framework contain the methods, properties, and enumerations you use to deliver derived credentials to the MobileIron client.

TABLE 54. DERIVED CREDENTIAL HEADER FILES

Header file	Description
ACDerivedCredential.h	Defines the ACDerivedCredential object that contains the certificates that comprise the derived credential. Your app sends an ACDerivedCredential object to the MobileIron client.
AppConnectDerivedCredentialService.h	Defines the ACDerivedCredentialService object which you use to: • Check if the MobileIron client is installed and supports derived credentials. • Check if sending derived credentials is currently allowed. • Inform the MobileIron client about the custom URL scheme to use to communicate to your app. • Send a derived credential to the MobileIron client.

Before adding derived credentials code

Before adding code to your app to send derived credentials to the MobileIron client, do the following tasks:

- Making your app an AppConnect app
- Declaring the appConnectdc URL scheme as allowed
- Registering as a handler of a URL scheme you define

Making your app an AppConnect app

Your app must be an AppConnect app and therefore must implement the AppConnect APIs to handle:

- AppConnect authorization
- AppConnect data loss prevention policies if applicable
- Dual mode behavior
- App-specific configuration if applicable

Therefore, follow the instructions in:

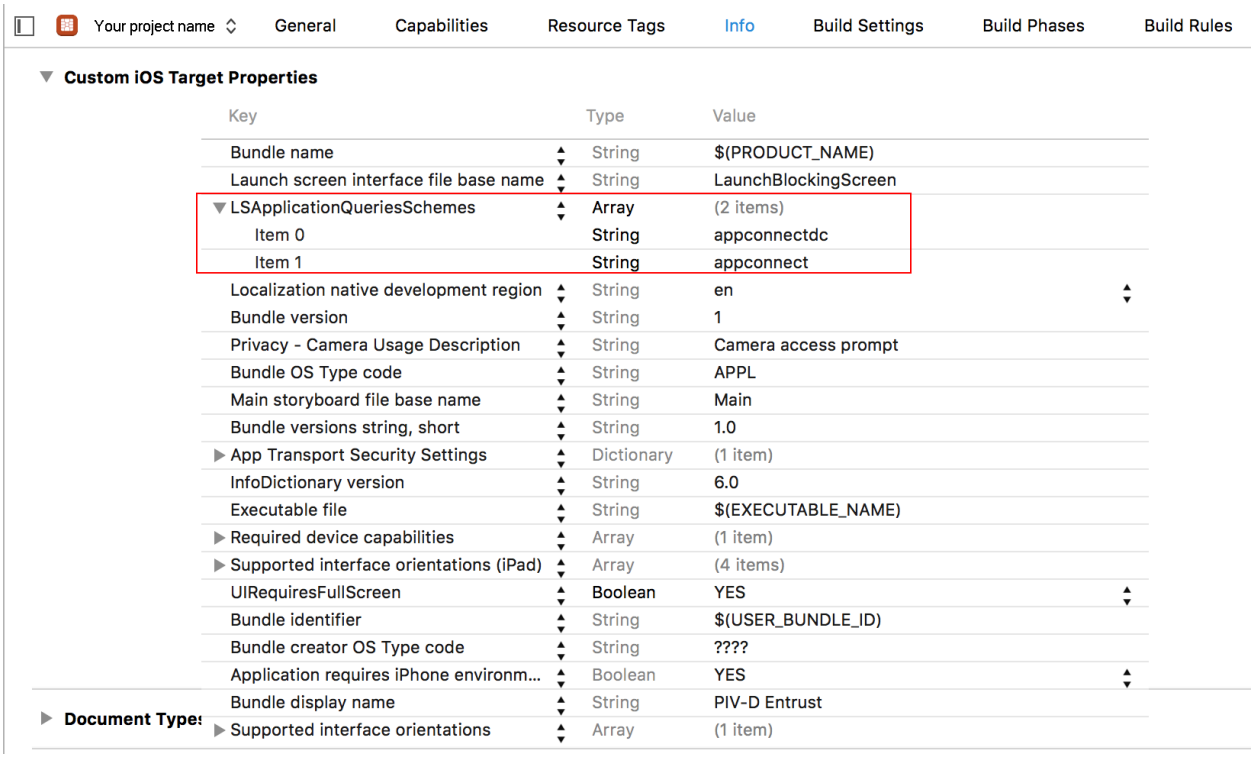
- Getting started tasks to set up your app to use the AppConnect library.
- Derived Credential Handling to handle AppConnect app authorization, data loss prevention policies, and app-specific configuration
- Developing Third-party Dual-mode Apps to make your app choose to behave in AppConnect mode (managed by MobileIron) or non-AppConnect mode.

Declaring the appConnectdc URL scheme as allowed

Declare the `appconnectdc` URL scheme in your app's Info.plist as an allowed URL scheme. Your app's instance of the AppConnect library uses the `appconnectdc` URL scheme to communicate with the MobileIron client.

To allow the `appconnectdc` URL scheme, add an item to the key called `LSApplicationQueriesSchemes`, which you already created to contain an item for the `appconnect` URL scheme. Add an item named `appconnectdc`.

The following screenshot from Xcode 7.3.1 illustrates the `appconnectdc` and `appconnect` items.

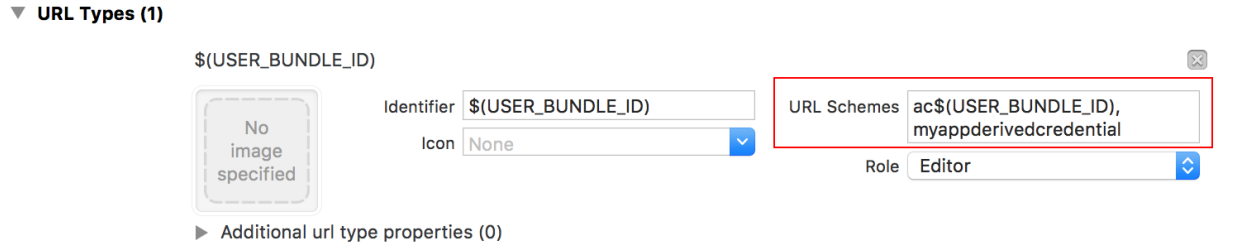


Registering as a handler of a URL scheme you define

The MobileIron client uses a custom URL scheme that your app defines to communicate with your app about derived credentials. Specifically, the MobileIron client can send a request to your app to create a new derived credential. The scheme the MobileIron client uses is:

```
<URL scheme you define>://new
```

Register your URL scheme by modifying the app's Info.plist, illustrated in the following Xcode 7.3.1 screenshot.



Sending derived credentials to the MobileIron client

Sending a derived credential to the MobileIron client requires the following tasks in your app:

- Handling the custom URL scheme in your app delegate
- Checking if the MobileIron client supports derived credentials
- Checking if sending credentials to MobileIron client is currently allowed
- Getting a derived credential
- Preparing a certificates array
- Preparing an ACDerivedCredential object
- Creating an ACDervicedCredentialService object
- Sending the certificates to the MobileIron client
- Handling secure services becoming available

Handling the custom URL scheme in your app delegate

You registered as a handler of a custom URL scheme that the MobileIron client uses to send your app derived credential requests. How to register as a handler is described in Registering as a handler of a URL scheme you define.

Add code to handle the custom URL scheme in your application delegate.

Objective-C example

```
-(BOOL)application:(UIApplication *)app openURL:(NSURL *) url
       options:(NSDictionary<NSString*,id> *) options {

       // If the URL is your app's custom URL scheme for receiving derived credential
       // communications from the MobileIron client (in this example "myappderivedcredential"),
       // and the command is "new"...

       if ( [url.scheme isEqualToString:@"myappderivedcredential"]
              && [url.host isEqualToString:@"new"] ) {

              // begin the logic for creating a new derived credential.
       }
}
```

Swift example

```
func application(_ app: UIApplication, open url: URL,
                 options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {

       // If the URL is your app's custom URL scheme for receiving derived credential
       // communications from the MobileIron client (in this example "myappderivedcredential"),
       // and the command is "new"...

       if (url.scheme == "myappderivedcredential" && url.host == "new") {

              // begin the logic for creating a new derived credential.
       }

       return true
}
```

Checking if the MobileIron client supports derived credentials

Before beginning the logic to create a derived credential and send it to the MobileIron client, check whether a MobileIron client is installed that supports derived credentials.

Header file: ACDerivedCredentialService.h

Method:

```
+(ACDerivedCredentialServiceSupport)derivedCredentialSupport;
```

In a typical call flow, the app does not call this method until:

- after the app is authorized (the `authState` property on the AppConnect object is `ACAUTHSTATE_AUTHORIZED`)
- after the app is in AppConnect Mode (the `managedPolicy` property on the AppConnect object is `ACMANAGEDPOLICY_MANAGED`)

Return values:

The method returns a value from the enumeration ACDerivedCredentialServiceSupport:

```
typedef NS_ENUM (NSInteger, ACDerivedCredentialServiceSupport) {
       ACDerivedCredentialServiceSupportPresent = 0,
       ACDerivedCredentialServiceSupportOldClient,
       ACDerivedCredentialServiceSupportMissingClient
}
```

Example:

```
ACDerivedCredentialServiceSupport supportStatus =
       [ACDerivedCredentialService derivedCredentialSupport];

   if (ACDerivedCredentialServiceSupportOldClient == supportStatus)
   {
       // Notify user to upgrade MobileIron client app to latest version.
       // The version running does not support derived credentials.
   }
   else if (ACDerivedCredentialServiceSupportMissingClient == supportStatus)
```

```
    {
        // Typically, this case won't happen if the app does not call +derivedCredentialSupport
        // until after the app is in AppConnect mode.
    }
    else
    {
        // Continue with code to check if sending credentials to the MobileIron client
        // is currently allowed.
}
```

Checking if sending credentials to MobileIron client is currently allowed

Before your app obtains derived credentials and prepares them for delivery to the MobileIron client, make sure sending credentials is currently allowed. It is allowed only if secure services are available and the MobileIron client supports receiving derived credentials. At this point, you have already verified that the MobileIron client supports derived credentials, but secure services are not necessarily available.

Header file: ACDerivedCredentialService.h

Method:

```
+(BOOL) canSendCredential;
```

Return values:

Returns `YES` if a MobileIron client that supports derived credentials is installed **and** secure services are available. Otherwise, returns `NO`.

Example:

```
if (![ACDerivedCredentialService canSendCredential])
{
       // Notify user that derived credentials cannot be obtained at this time.
       // When secure services become available, the AppConnect library calls
       // the notification method -appConnect:secureServicesAvailabilityChangedTo:.
       // At that time, the app can continue with the logic to get derived credentials
       // and deliver them to the MobileIron client.
}
else
{
       // Continue with code to get derived credential.
}
```

Getting a derived credential

Your app gets a derived credential only after both of the following are true:

- Your app has determined that derived credentials are supported (`+derivedCredentialSupport`)
- Your app has determined that it is allowed to send derived credentials to the MobileIron client at this time (`+canSendCredential`)

Your app gets the certificates that comprise the derived credential according to its own requirements.

Preparing a certificates array

After your app has obtained the certificates that comprise the derived credential, prepare an NSArray of the certificates. Each array entry is an NSDictionary object. The following table describes each entry in the NSDictionary object.

TABLE 55. ENTRIES IN THE NSDICTIONARY OBJECT OF A CERTIFICATE ARRAY ENTRY

Entry	Key	Value
The certificate tag	`ACDerivedCredentialPayloadKeyTag`	An NSString object The value describes the expected use of the certificate. The MobileIron client uses the value to determine which certificates to deliver to an AppConnect app. The value is one of the certificate tags defined in ACDerivedCredential.h: • `ACDerivedCredentialTagAuthentication` • `ACDerivedCredentialTagSigning` • `ACDerivedCredentialTagEncryption` • `ACDerivedCredentialTagEscrow` You can use each value in only one NSDictionary object in the NSArray. That is, you can associate each value with only one certificate in the derived credential.
The certificate contents	`ACDerivedCredentialPayloadKeyCert`	An NSData object The object contains the DER-encoded certificate data.
The certificate's private key	`ACDerivedCredentialPayloadKeyPrivateKey`	An NSData object The object contains the DER-encoded private key of the certificate. The private key must be in PKCS #8 format.

Header file: ACDerivedCredential.h -- contains definitions of constants

Example:

```
#import <AppConnect/ACDerivedCredential.h>

NSData *certificateData;       // contains DER-encoded certificate data
NSData *privateKeyData;        // contains DER-encoded private key, in PKCS #8 format.

// Insert code that gets the certificate used for authentication and its private key.

NSDictionary *authCertificatePackage = @{
   ACDerivedCredentialPayloadKeyTag : ACDerivedCredentialTagAuthentication,
   ACDerivedCredentialPayloadKeyCert : certificateData,
   ACDerivedCredentialPayloadKeyPrivateKey : privateKeyData
};

// Insert similar code for populating NSDictionary objects with certificates to be used for
// encryption or signing. The number of NSDictionary objects you populate depends on the
// number of different uses for certificates your app supports.

// Place the NSDictionary entries into an array
NSArray *certificatesArray =
       @[authCertificatePackage, encryptCertificatePackage, signingCertificatePackage];
```

Preparing an ACDerivedCredential object

After you have prepared the certificates array, create and initialize an ACDerivedCredential object.

Header file: ACDerivedCredential.h

Method:

```
-(instancetype)initWithName:(NSString *)name
       serialNumber:(NSString *)serialNumber
       expirationDate:(NSDate *)expirationDate
       certificates:(NSArray *)certificates NS_DESIGNATED_INITIALIZER;
```

TABLE 56. PARAMETERS FOR -INITWITHNAME:SERIALNUMBER:EXPIRATIONDATE:CERTIFICATES:

Parameter	Description
`name`	A human readable name for this derived credential payload. The MobileIron client displays this name with the derived credential information.
`serialNumber`	A unique identifier for the derived credential. Typically, this serial number is provided to your app by your derived credential provider.
`expirationDate`	The expiration date of the derived credential. This date is not necessarily the same as the expiration date of each certificate in the derived credential. It is the responsibility of the derived credential provider to enforce this expiration date according to the provider's requirements.
`certificates`	The array of certificates that comprise the derived credential.

Return values:

- Returns an ACDerivedCredential object if no errors occur.
- Returns `nil` if:
  - the `name` or `serialNumber` parameter is `nil` or an empty string
  - the `expirationDate` parameter is `nil`.
  - the `certificates` parameter is `nil` or an empty array

NOTE: The AppConnect library within the app logs an error to the device's console when this method returns nil.

Example:

```
ACDerivedCredential *derivedCredential =
       [[ACDerivedCredential alloc] initWithName:@"Derived Credential Name"
                                    serialNumber:@"123-4567-8910"
                                    expirationDate:expirationDate
                                    certificates:certificatesArray];
```

Creating an ACDervicedCredentialService object

Create an ACDerivedCredentialService object for communicating with the MobileIron client.

Header file: ACDerivedCredentialService.h

Method:

```
-(instancetype)initWithBrand:(NSString *)brand
       callbackScheme:(NSString *)callbackScheme NS_DESIGNATED_INITIALIZER;
```

TABLE 57. PARAMETERS FOR -INITWITHBRAND:CALLBACKSCHEME:

Parameter	Description
`brand`	The name of the derived credentials provider.
`callbackScheme`	The custom URL scheme that your app defines for the MobileIron client to communicate with your app about derived credentials.

Return values:

- Returns an ACDerivedCredentialService object if no errors occur
- Returns `nil` if either of the following are true:
  - any parameter is `nil` or an empty string
  - the MobileIron client does not support derived credentials

NOTE: The AppConnect library within the app logs an error to the device's console when this method returns nil.

Example:

```
ACDerivedCredentialService *derivedCredentialService =
       [[ACDerivedCredentialService alloc] initWithBrand:@"Derived Credential Provider Name"
                callbackScheme:@"MyCustomDcScheme"];
```

Sending the certificates to the MobileIron client

After creating the ACDerivedCredential and ACDerivedCredentialService objects, send the derived credential to the MobileIron client.

Header file: ACDerivedCredentialService.h

Method:

```
-(BOOL)sendDerivedCredential:(ACDerivedCredential*)derivedCredential withError:(NSError **)er-
ror;
```

TABLE 58. PARAMETERS FOR -SENDDERIVEDCREDENTIAL:WITHERROR:

Parameter	Description
`derivedCredential`	The ACDerivedCredential object you created and initialized.
`error`	A reference to an NSError pointer. If an error occurs, the method returns `NO and` updates the pointer to point to an NSError object describing the problem. Possible values of the NSError object's `code` property are defined in the enumeration ACDerivedCredentialServiceErrorCode. Although allowed, passing `NULL` for this parameter is not recommended, since the app's error handling would be limited.

Return values:

Returns `YES` if the certificates have been sent to the MobileIron client. Otherwise, returns `NO`.

NOTE: When the return value is YES, the MobileIron client is launched. Control does not automatically return to the app. Therefore, a typical behavior in this case is to change to a "home" screen that offers options for what the device user can do next.

Example:

```
NSError *error = nil;
BOOL credentialSent = [derivedCredentialService sendDerivedCredential:derivedCredential
                                                    withError:&error];

if (!credentialSent) {
       // Sending the derived credential to the MobileIron client failed.
       // Examine the error and handle appropriately, notifying the user as necessary.

       // If the error is ACDerivedCredentialErrorServiceUnavailable, notify the user
       // and wait for the callback method -appConnect:secureServicesAvailabilityChangedTo:
       // to indicate that secure services are available before trying again.
}
```

```
else {
       // The derived credential was successfully sent to the MobileIron client.
}
```

Handling secure services becoming available

Sending derived credentials to the MobileIron client requires AppConnect's secure services to be available. Your app calls the method `+canSendCredential,` which checks if secure services are available and the MobileIron client supports receiving them.

If `+canSendCredential` returns `YES`, your app proceeds with getting and delivering derived credentials. However, secure services could become unavailable before you deliver the derived credentials to the MobileIron client. In that case, your app must take the appropriate actions. If `+canSendCredential` returns `NO`, your app notifies the user and must wait for secure services to become available.

Therefore, implement the notification method `-appConnect:secureServicesAvailabilityChangedTo:` in the AppConnectDelegate protocol, defined in AppConnect.h:

```
-(void) appConnect:(AppConnect *)appConnect secureServicesAvailabilityChangedTo:
           (ACSecureServicesAvailability)secureServicesAvailability;
```

Example:

```
-(void)appConnect:(AppConnect *)appConnect secureServicesAvailabilityChangedTo:
       (ACSecureServicesAvailability)secureServicesAvailability {

       if (ACSECURESERVICESAVAILABILITY_AVAILABLE == secureServicesAvailability) {
              // Notify the user as necessary, according to your app state.
              // The app can now proceed to logic for getting and delivering derived credentials.
       }
       else {
              // Secure services are not available.
              // The app cannot deliver derived credentials to the MobileIron client.
              // Notify the user as necessary, and change your app state appropriately.
       }
}
```

AppConnect for iOS SDK Revision History

AppConnect 4.6.0 for iOS SDK revision history

This release provides the following:

New features summary

This release includes the following new features and enhancements:

- **Improvements to memory consumption**: Secure File I/O APIs have been optimized to decrease memory consumption while processing large files.
- **Two SDK variants**: Due to Apple deprecating the UIWebView class, the AppConnect for iOS SDK is available in two variants: one with UIWebView and WKWebView support, and another with WKWebView

support, but no UIWebView support. The AppConnect SDK without UIWebView support is provided for apps that will be submitted to the App Store.
See AppConnect for iOS SDK variants and AppConnect for iOS SDK contents.

- **Support for UIScene**: A new method `-sceneWillConnectToSessionWithOptions:` is provided to support apps using UIScene. If your application supports UIScene, call the method `-sceneWillConnectToSessionWithOptions:` when initializing the AppConnect library.
See Initialize the AppConnect library and UIScene support.

Resolved issues

This release includes the following new resolved issues:

- **AP-5422**: Fixed issue with tunneled requests authentication when app implements `URLSession:didReceiveChallenge:completionHandler:` method of the URLSessionDelegate protocol.
- **AP-5328**: Fixed an issue where AppConnect apps flipped to the MobileIron client app for password authentication. Now the passcode prompt is seen within the wrapped app.

AppConnect 4.5.3 for iOS SDK revision history

This release provides the following:

- Resolved issues

Resolved issues

This release provides the following new resolved issues in the SDK and wrapper:

- AP-5376, APG-1177: Fixed an issue where redirected server requests could fail to connect.

AppConnect 4.5.2 for iOS SDK revision history

This release does not provide any updates to the SDK.

AppConnect 4.5.1 for iOS SDK revision history

This release does not provide any updates to the SDK.

AppConnect 4.5.0 for iOS SDK revision history

This release provides the following:

Resolved issues

This release provides the following new resolved issues:

- AP-5256: Workaround for a bug in a third-party app security framework, which caused a crash when used with AppConnect.
- AP-5241: Fixed crash [ACAppInterfaceBus displayMessage:scheme:completion:].
- AP-5199: Sometimes AppConnect apps failed to unlock using biometric authentication if the device passcode was set as the fallback option. Users may have seen this issues if the Check-in interval and the AutoLock interval are small and equivalent. This issue is fixed.

Known issues

This release includes the following new known issues:

- APG-1154: UIScene apps, introduced in iOS 13, are not supported. The application lifecycle delegate methods are not called, so AppConnect is never initialized.

AppConnect 4.4.2 for iOS SDK revision history

This release provides the following:

Resolved issues

This release provides the following new resolved issues:

- AP-5245: Fixed a Secure File I/O thread-safety issue which could cause I/O errors when writing to multiple files simultaneously. Note that I/O to individual files should always be done from a single thread.
- AP-5253: Fixed an exception when launching apps in the Xcode Simulator.

Known issues

This release includes the following new known issue:

- AP-5252: Web@Work 2.9.0.0 for iOS with Chromium does not trust some sites. For more information, see the following Knowledge Base article in the MobileIron Community: Web@Work - Certain sites may not be trusted when using Chromium engine.

AppConnect 4.4.1 for iOS SDK revision history

This release provides the following:

Resolved issues

This release includes the following new resolved issues:

- AP-5233: Under certain conditions when adding cookies to a network request, the cookies were dropped after receiving an HTTP 302 redirect. This issue is fixed.

Known issues

This release includes the following new known issues:

- APG-1154: UIScene apps, introduced in iOS 13, are not supported. The application lifecycle delegate methods are not called, so AppConnect is never initialized.

AppConnect 4.4.0 for iOS SDK revision history

This release provides the following:

New features summary

This release includes the following new features and enhancements:

- **Support for iOS 13**: AppConnect apps work as expected on iOS 13 devices.
- **The -displayMessage method is updated:** The following method is deprecated:

  ```
  -(void)displayMessage:(NSString *)message;
  ```

  Instead, use the following new method:

  ```
  -(void)displayMessage:(NSString *)message withCompletion:(void(^)(BOOL
  success))completion;
  ```

  The native sample apps included with the SDK are updated.
- **armv7s architecture**: Support for the armv7s architecture has been dropped.

Resolved issues

This release provides the following new resolved issues:

- AP-5158: iOS 13 changed the identification for iPad devices. If your iPad is upgraded to iOS 13, MobileIron recommends that you also upgrade to MobileIron Core to one of the following patch releases: 10.2.0.2, 10.3.0.2, or 10.4.0.1. These patches contain the fixes for the changes in iOS 13 for iPad identification.
- AP-5179: On devices running iOS 13, openURL does not return the bundle ID of the calling app if the team ID is not the same. This issue is fixed with AppConnect 4.4.0 for iOS. To address the issue, update to AppConnect 4.4.0.
- AP-5201: Previously, the NSProxy instance proxying application delegate did not receive application lifecycle callbacks. This issue is fixed.
- AP-5207: On devices running iOS 13, AppConnect apps can Open files to other apps when Open In is disabled. This issue is fixed with AppConnect 4.4.0 for iOS. To address the issue, update to AppConnect 4.4.0.
- AP-5166: On devices running iOS 13, NSURLSession failed. This issue is fixed with AppConnect 4.4.0 for iOS. To address the issue, update to AppConnect 4.4.0.
- AP-5169: On devices running iOS 13, Email+ for iOS displayed a black background in app switcher. This issue is fixed with AppConnect 4.4.0 for iOS. To address the issue, update to AppConnect 4.4.0.
- AP-5174: Fixed the root cause due to which Email+ for iOS crashed intermittent.
- AP-5206: Previously, the AppConnect for iOS SDK was not calling applicationDidBecomeActive. This issue is fixed.

Limitations

This release includes the following new limitations:

- AP-5186: The openURL API in iOS 13 provides the bundle ID of the calling app only if the calling app has the same team ID. Due to this limitation, the Open From feature does not work on iOS 13 devices.
- AP-5164: Sharing files with the Chrome extension if Open In is restricted may cause the application to freeze.
- AP-5159: On devices running iOS 13, the "Unable to Share Document with selected application" prompt is not shown unless the Share dialog is closed.

AppConnect 4.3.1 for iOS SDK revision history

This release does not provide any new features.

Support for the armv7s architecture is deprecated.

Resolved issues

This release provides the following new resolved issue:

- APG-1132: Fixed a potential crash in the NSURLSession delegate_task:didCompleteWithError: method.

AppConnect 4.3.0 for iOS SDK revision history

New features

- **Support for MobileIron AppStation**
  Apps built with the AppConnect 4.3.0 for iOS SDK can run with MobileIron AppStation as the MobileIron client app instead of MobileIron Go. Administrators can use MobileIron AppStation on devices which are interacting with a MobileIron Cloud tenant that supports Mobile Apps Management (MAM) but not Mobile Device Management (MDM).
  For your AppConnect app to support AppStation:
  - Declare the `alt-appconnect`URL scheme in your app's Info.plist as another allowed URL scheme.
    See Declare the AppConnect URL schemes as allowed.
  - Rebuild your app with the AppConnect 4.3.0 for iOS SDK.
    See Task lists for upgrading the SDK in your app.
- **Support for Open From data loss prevention policy**
  The AppConnect 4.3.0 for iOS SDK adds support for the Open From data loss protection policy. Although the AppConnect library enforces the policy as configured on the MobileIron server, apps can implement methods that allow them to inform the end user about the policy. For details, see Open From policy API details .

At the date of this AppConnect release, no MobileIron servers support this policy.

- **iOS 9 no longer supported**
  AppConnect 4.3.0 for iOS is not supported on iOS 9 devices.
  See Product versions required .

AppConnect 4.2.1 for iOS SDK revision history

New features

- **Allow AppConnect apps to send custom cookies in web requests**
  Some web pages inject custom cookies into web requests. For example, when an end user taps on a link in a web page, the page's JavaScript injects a custom cookie. If a user makes such a request from a web page displayed in an AppConnect app, by default AppConnect does not include the injected cookies in the web request, which can cause the request to fail. AppConnect now includes the custom cookies in the request if the MobileIron server administrator includes the following key in the app's app-specific configuration on the MobileIron server: `MI_AC_USE_ORIGINAL_COOKIES_FOR_DOMAINS`. The value of the key is a comma-separated string listing the domains for which the custom cookies should be included. Make sure no spaces are included in the value.
  For example:
  `www.somewebsite.com,somename.someotherwebsite.com`

Limitations

- AP-5026: A Xamarin app crashes if it uses custom code to copy text rather than the native iOS copy functionality.

AppConnect 4.2 for iOS SDK revision history

New features

- Added support for escrow certificates for apps that use the derived credentials APIs to deliver derived credentials to the MobileIron client. Note that MobileIron support for this feature will be available only when all involved MobileIron products support this feature.
  See Sending derived credentials to the MobileIron client.

Resolved issues

- AP-4919: Fixed an issue that caused an AppConnect app to crash when it used the same object as a delegate for multiple UI elements.
- AP-4150: After an AppConnect SDK or Cordova app became inactive and the AppConnect library blurred the screen, a noticeable delay occurred when removing the blur when the app became active. This issue has been fixed.

Known issues

- AP-4940: The LookUp option in the iOS context menu allows data to be shared to non-AppConnect apps regardless of the **Open In** and **Copy/Paste To** data loss prevention policies.

AppConnect 4.1.1 for iOS SDK revision history

This AppConnect release has no new features.

Resolved issues

- AP-4920: When an AppConnect's app upload request is redirected, the request failed when using AppTunnel. This issue has been fixed by converting the stream request to a body request when using AppTunnel. Note that you can override the conversion by adding a key-value pair to the app's AppConnect configuration. Add MI_AC_DISABLE_HTTP_STREAM_CONVERSION with the value Yes.
- AP-4917: Fixed compilation issues when using the AppConnect for iOS SDK with projects containing Objective-C++ files.
- APG-1118: Fixed an issue where apps subclassing NSProxy could crash on launch with the error `-[NSProxy doesNotRecognizeSelector:_ACDecoratorClass]`.
- APG-1097: Provides a workaround to a known bug in NSURLSession that sometimes causes the form body to be missing in connections in AppConnect apps when using AppTunnel.

Known issues

- AP-4919: If an AppConnect app uses the same object as a delegate for multiple UI elements, the app crashes.

AppConnect 4.1 for iOS SDK revision history

This AppConnect release has several new features. It has no new known or resolved issues or limitations.

New features

Certificate pinning support

This AppConnect release supports certificate pinning for AppConnect apps to heighten security for communication between AppConnect apps and enterprise servers or cloud services.

Using certificate pinning requires:

- Configuration on the MobileIron server.
  For MobileIron Core, see “Certificate pinning for AppConnect apps” in the MobileIron Core AppConnect and AppTunnel Guide.
- Mobile@Work 10.0.0.0 for iOS through the most recently released version as supported by MobileIron.

This feature requires no additional development in the app.

Lock AppConnect apps when screen is off

This AppConnect release supports automatically logging out device users from AppConnect apps when the device screen is turned off due to either inactivity or user action.

This feature requires:

- Configuration on the MobileIron server.
  For MobileIron Core, see “Configuring the AppConnect global policy” in the MobileIron Core AppConnect and AppTunnel Guide.
- Mobile@Work 10.0.0.0 for iOS through the most recently released version as supported by MobileIron.

This feature requires no additional development in the app.

Overriding the Open In Policy for openURL: with the mailto: scheme

This AppConnect release allows either the app or MobileIron server administrator to override the Open In policy when the policy blocks the iOS native email app when the app calls `openURL:` with the `mailto:` scheme.

The AppConnect library overrides the Open In policy for native email if either of the following are true:

- The MobileIron server administrator added the key MI_AC_DISABLE_SCHEME_BLOCKING with the value `true` to the app’s app-specific configuration.
- The app’s Info.plist contains the MI_APP_CONNECT dictionary with the key MI_AC_DISABLE_SCHEME_BLOCKING with the value `YES`.

NOTE: THE MI_APP_CONNECT dictionary is new in this release.

See Open In policy API details .

SwiftFileSharing demonstrates sharing secure files from an extension

This AppConnect release has enhanced the SwiftFileSharing sample app to demonstrate how to share secure files from an app's extension using AppConnect APIs.

See Sharing secure files from an extension.

AppConnect 4.0 for iOS SDK revision history

New features

iOS 8 no longer supported

AppConnect 4.0 for iOS is not supported on iOS 8 devices.

See Product versions required .

Dynamic frameworks

The AppConnect 4.0 for iOS SDK changes the AppConnect.framework from a static to dynamic framework. Therefore, to upgrade an app that used a previous AppConnect SDK, or to incorporate the SDK for the first time into your app, see Getting Started with the AppConnect for iOS SDK.

One of the necessary steps in using the dynamic AppConnect.framework is to remove extra architectures from the AppConnect app's binary. Removing desktop architectures is required before submitting your app to the Apple App Store. The AppConnect for iOS SDK provides a script for this purpose. The script is called post_embed_ actions.sh.

Also, as part of the reorganization relating to dynamic frameworks, AppConnect.h is now an umbrella header which imports all the AppConnect.framework headers. AppConnectInterface.h now contains the definitions of the AppConnect interface and the AppConnectDelegate protocol. See Header files in AppConnect.framework.

Swift support

The AppConnect 4.0 for iOS SDK supports Swift apps. See Using the AppConnect framework in a Swift app.

Secure file sharing from an extension

An AppConnect app can now provide an app extension, specifically a Document View Controller extension, to share secure files with other AppConnect apps. A file can be shared with all AppConnect apps or with only specific AppConnect apps. The files that the extension shares must be secure files, written with the secure file I/O APIs.

See Sharing secure files from an extension.

Drag and Drop data loss prevention policy support

MobileIron server administrators can set a drag and drop policy for each AppConnect app. It specifies whether AppConnect apps can drag content to all other apps, to only other AppConnect apps, or not at all.

The AppConnect library enforces this policy. When the policy allows dragging content to only other AppConnect apps, the AppConnect library notifies your app when the device user attempts to drag content to a non-AppConnect app. Your app can then notify the device user of the situation. Your app provides no other code to support the drag and drop policy.

NOTE: This feature is not supported with MobileIron Cloud.

See:

- Drag and drop policy API details
- Test data loss prevention policy handling

New callback method -openURLAttemptedWhenUnauthorizedForURL:

A new callback method `-openURLAttemptedWhenUnauthorizedForURL:` is provided. This method is called when the app attempts to call `-openURL:` with the `mailto` scheme but no app that can handle the scheme is allowed by the Open In DLP policy.

See Open In policy API details .

Native email control using the Open In DLP policy

The Open In Data Loss Prevention policy now includes controlling whether an app can share documents with the native iOS mail app. Opening a document with the native iOS mail app is allowed only if one of the following is true:

- Open In is allowed for all apps

- Open In is allowed for only whitelisted apps, and the native iOS mail app is in the whitelist. The whitelist must contain both of these bundle IDs: com.apple.UIKit.activity.Mail and com.apple.mobilemail.

Additionally, the new callback method `-openURLAttemptedWhenUnauthorizedForURL:` is called when the app attempts to call `openURL:` with the `mailto:` scheme, and one of the following is true:

- Open In is not allowed, and Email+ for iOS is not installed on the device.
- Open In is allowed only for Secure Apps, and Email+ is not installed on the device.

NOTE: In both of the above cases, if Email+ for iOS is installed on the device, it is launched.

See Open In policy API details .

App extension control using the Open In DLP policy

The Open in data loss protection policy now includes restricting access to the iOS extensions that apps provide. Specifically:

Open In DLP for host app (the app using the extension)	Extension behavior
All apps allowed	The host app can use any app's extension for Open In.
Only AppConnect apps allowed	The host app can use only extensions provided by AppConnect apps for Open In.
Whitelist	The host app can use only extensions of apps in the whitelist for Open In.

This addition has no impact on your app's implementation of the Open In DLP APIs.

Custom keyboard use controlled by MobileIron server

The MobileIron server can now control custom keyboard use by your AppConnect app. If the administrator does not configure this choice, your app can choose to reject custom keyboard use.

See Custom keyboard control.

Screen blurring

AppConnect 4.0 for iOS adds support for blurring screens when the app becomes inactive. If your app provided its own screen blurring, remove that code. By using the AppConnect library's screen blurring capability, all AppConnect apps behave consistently.

To enable screen blurring, add the key MI_AC_PROVIDE_SCREEN_BLUR to your app's Info.plist as a Boolean. Set the value to YES.

When you set the Info.plist key MI_AC_PROVIDE_SCREEN_BLUR to YES, the MobileIron server administrators can disable screen blurring by setting a key-value pair on the server for your app's configuration. The server key is MI_AC_ENABLE_SCREEN_BLURRING with the value false.

See Add AppConnect-related entries to your Info.plist.

Requirement for Face ID usage Info.plist entry

Include **Privacy - Face ID Usage Description** to your app's info.plist, with a string value indicating the purpose of Face ID use. For example, add the value **AppConnect**. If you manually add this key, its name is NSFaceIDUsageDescription.

Server administrators can allow the use of Touch ID or Face ID instead of an AppConnect passcode. Therefore, this Info.plist entry is required on iOS 11 through the most recently released version as supported by MobileIron.

Support for sending AppConnect logs from Mobile@Work

AppConnect apps using AppConnect 4.0 for iOS support the feature in Mobile@Work for iOS that sends AppConnect logs to an email address of your choice, such as a company's helpdesk. This feature requires Mobile@Work 9.8 for iOS through the most recently released version as supported by MobileIron.

Mobile@Work displays the option to send logs on the app's status details screen, available in Mobile@Work at **Settings > Secure Apps > *<app name>*.** The option is at the bottom of the screen with this text: **Send *<app name>* Logs.**

The option is displayed only for AppConnect apps using AppConnect 4.0 for iOS However, the displayed option is disabled if the app's AppConnect authorization status is not authorized.

When the option is displayed and enabled, tapping it brings up the list of apps able to share the log files, such as email apps, if you included the following key-value pair for the app in its AppConnect app configuration:

- **MI_AC_ENABLE_LOGGING_TO_FILE** set to **Yes**

Securing sensitive data such as encryption keys

AppConnect 4.0 for iOS adds classes to provide heightened security for especially sensitive data, such as encryption keys and passwords. These classes use hardware capabilities (Apple's Secure Enclave) to reduce the sensitive data's attack surface, because the data is never stored in plain-text in memory.

See Securing sensitive data such as encryption keys.

New category ACFileHandle (ACSharedSecureData)

Use the new category ACFileHandle (ACSharedSecureData) in addition to the existing category NSData (ACSharedSecureData) if you want to encrypt the data that your app stores **and** you want the app to share the data with another AppConnect app.

See Secure file I/O API details.

New custom cryptography methods

The AppConnect 4.0 for iOS SDK has deprecated the following methods:

```
-(NSError *)derivedAppKey:(uint8_t *)pKey withIndex:(NSString *)index;
```

```
-(NSError *)derivedSharedKey:(uint8_t *)pKey withIndex:(NSString *)index;
```

New methods are available that each return an ACSensitiveData object. If you are upgrading your app to use the AppConnect 4.0 for iOS SDK, MobileIron recommends you use the new methods to take advantage of the features of ACSensitiveData class.

See Encryption keys for custom cryptography and Securing sensitive data such as encryption keys.

Automatic policy status updates sent to MobileIron server

The AppConnect library now automatically sends a status update to the MobileIron server when it receives the following changes:

Change	Status update that AppConnect library sends to MobileIron server
Open In policy	Informs server that the policy change has been applied.
Pasteboard policy	Informs server that the policy change has been applied.
Print policy	Informs server that the policy change has been passed to the app.
Configuration values	Informs server that the configuration change has been passed to the app.
Authentication status	Informs server that the authentication change has been passed to the app.

This change has no impact on your app's implementation. Your app should continue to always call the appropriate notification acknowledgment method:

```
-authStateApplied:message:
```

```
-configApplied:message:
```

```
-openInPolicyApplied:message:
```

```
-pasteboardPolicyApplied:message:
```

```
-printPolicyApplied:message:
```

```
-secureFileIOPolicyApplied:message:
```

Resolved issues

- AP-4324: The following methods now return an empty NSData object, instead of nil, if EOF was reached:

```
- (NSData *)availableData;
- (NSData *)availableDataWithError:(NSError *__autoreleasing *)error;
- (NSData *)readDataToEndOfFile;
- (NSData *)readDataToEndOfFileWithError:(NSError *__autoreleasing *)error;
- (NSData *)readDataOfLength:(NSUInteger)length;
- (NSData *)readDataOfLength:(NSUInteger)length error:(NSError *__autoreleasing *)error;
```

- AP-4202: Custom protocol classes set to NSURLSessionConfiguration were previously ignored in AppConnect apps. This issue has been fixed.
- AP-4133: Added ability to use NSURLConnection with NSURLSession networking with AppTunnel.

Known issues

- AP-4657: The "unauthorized message" screen is blurred. It continues to be blurred until the next time the app switches to the MobileIron client app. After the next AppConnect checkin, the screen is no longer blurred.

Limitations

- AP-4720: On some devices, screen blurring does not occur when going to the Task Switcher.

AppConnect 3.5 for iOS SDK revision history

New features

iOS 11 compatibility

This version of the AppConnect for iOS SDK is compatible with devices running iOS 11 Beta 7. At the time of this AppConnect release, the GA version of iOS 11 is not available.

IMPORTANT: Upgrade your app to use AppConnect 3.5 for iOS for your app to run on iOS 11 devices. Apps built with SDK versions prior to 3.1.3 crash on iOS 11 devices. Apps built with version 3.1.3 do not crash, but the AppConnect library does not handle the pasteboard data loss prevention policy correctly.

For more information, see Product versions required on page 33.

Open In changes

- A new optional callback method called `-appConnect:openInAttemptedWhenACOpenInPolicyBlocked:` has been added in the AppConnectDelegate protocol.
  See The -appConnect:openInAttemptedWhenACOpenInPolicyBlocked: callback method on page 94.
- Because of iOS implementation changes, if an app uses UIActivityViewController to execute Open In, when the Open In policy specifies a whitelist, Open In to *all* apps is not allowed. Therefore, use only UIDocumentInteractionController to execute Open In.
  See Overview of Open In handling on page 91.
- The AppConnect library supports a new key-value pair from the MobileIron server that tells the library not to enforce the Open In policy. A MobileIron server administrator determines if this behavior is appropriate for an enterprise. An app makes no changes relating to this feature.
  See “Overriding the Open In Policy for the app” in the administrator documentation *MobileIron Core MobileIron Core AppConnect and AppTunnel Guide*.

Sample app Xcode projects now compatible with Xcode 8.3

The Xcode projects for the sample apps HelloAppConnect and DualMode are now compatible with Xcode 8.3. They were previously compatible with Xcode 6.4.

See AppConnect for iOS SDK contents on page 24.

Resolved issues

- AP-4145: URL requests made on a background thread were not tunneled if the AppConnect library in the app had not received the AppTunnel rules. The issue has been fixed because the AppConnect library now blocks URL requests until after it has received the AppTunnel rules.
- AP-3917: When a URL request using NTLM authentication was tunneled with AppTunnel, an error occurred when the device user was prompted with the user credentials dialog. The dialog displayed the Standalone Sentry host name instead of the URL request's host name. The issue has been fixed.

Limitations

- AP-4302: Apps that use UIDocumentInteractionController's preview API will not be able to share documents with other apps, because iOS 11 beta 6 and 7 allow sharing only with certain built-in extensions.

Releases prior to AppConnect 3.5 for iOS SDK revision history

For the revision history of releases prior to AppConnect 3.5 for iOS SDK, see the"MobileIron AppConnect 4.2 for iOS SDK App Developers Guide", available on https://help.mobileiron.com.