VMware AirWatch SDK for iOS (Swift) Technical Implementation Guide

Empowering your enterprise applications with MDM capabilities

VMware AirWatch SDK for iOS (Swift) v17.11

Have documentation feedback? Submit a Documentation Feedback support ticket using the Support Wizard on support.air-watch.com.

Table of Contents

Chapter 1: Overview

Introduction to the SDK for iOS (Swift)

The AirWatch Software Development Kit (SDK) is a set of tools allowing organizations to incorporate a host of features and functionality into their custom-built iOS applications. The AirWatch SDK enhances the security and functionality of those applications and in turn helps save application development time and money. This document reviews the SDK implementation process, and also covers the options that are available and how they are configured.

Version of AirWatch SDK for iOS and AirWatch Console

- AirWatch SDK for iOS v17.11
- AirWatch Console v9.1.1+

Supported iOS Components

- Apple iOS - 9.0+
- Xcode - 9.1 and 9.2
- Code base
  - Swift
    - 3.1
    - 3.2.2
    - 4.0.2
  - Objective-C 2.0+

Swift Examples Apply to Objective-C

This release of the SDK was developed to include applications coded in both Objective-C and Swift. The examples are in Swift. However, the setup process and the function names are the same. Use Objective-C syntax to setup the SDK functions.

Chapter 2:
AirWatch SDK for iOS (Swift) Installation

Migrate the AirWatch SDK for iOS Objective-C Version to the Swift Version

To migrate to a version of the AirWatch SDK for iOS for Swift, remove the old SDK and add the current one to your environment.

See Component Changes in the AirWatch SDK for iOS on page 7 for changes to make to your project to prevent build errors.

Remove the Objective-C Version of the SDK

Delete the listed AirWatch frameworks and libraries to remove the SDK.

1. On the **General** tab in your project, delete the **AWSDK.framework** from both the **Embedded Binaries** and **Link Framework and Libraries** areas.
2. Open the **Build Phases** tab in the project settings of your application.
3. Delete **AWKit** from your project.
4. Delete **AWlocalization** from your project.

Add the Swift Version of the SDK

Add AirWatch frameworks and edit the locations of the listed calls to migrate SDK behaviors to the current version. If you do not edit the listed call locations, the UI behavior is inconsistent with the previous SDK version.

1. Drag and drop the current **AirWatchSDK** framework and the **AWCMWrapper** file into your **Link Binary with Libraries** step in the build phase section of your project settings.
2. Change the location of your **StartSDK** call. Call it in the **didFinishLaunchingWithOptions** method that is inside your application delegate class.

   In versions before the SDK v17.x, you called **awcontroller.start()** within the **applicationDidBecomeActive** method.
3. Build your project.
4. Resolve naming differences and API differences that changed in the new SDK causing build errors.

Share Your Keychain

Share your keychain between the SDK applications so you can use all the SDK capabilities. See Keychain Access Group Entitlements on page 17.

Component Changes in the AirWatch SDK for iOS

If you migrate an older version of the SDK to install it, review the list of changed components. Update names and locations of components to prevent or resolve build errors caused by the differences between SDK versions.

Samples present the old version of the code followed by the current code.

Component	Sample Code

AWController start
In the previous SDK you called **awcontroller.start ()** within the **applicationDidBecomeActive** method.
In the current SDK, start the SDK within the **didFinishLaunchingWithOptions** method inside your application delegate class.
You will get inconsistent UI behaviors from the SDK if you do not make this change.
	`///5.9.X Implementation`
	`func applicationDidBecomeActive(_ application: UIApplication)`
	`let awc = AWController.clientInstance()`
	`awc.delegate = self`
	`awc.callbackScheme = "myAppName"`
	`awc.start()`
	`}`
	
	`///17.X Implementation`
	`func application(_ application: UIApplication, didFinishLaunchingWithOptio`
	`launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {`
	`let awc = AWController.clientInstance()`
	`awc.delegate = self`
	`awc.callbackScheme = "myAppName"`
	`awc.start()`
	`return true`
	`}`

CanhandleProtectionSpace (Integrated Authentication)
Update the code for authentication challenges and chain validation.
	`///5.9.X Implementation`
	`try AWController.clientInstance().canHandle(challenge.protectionsSpace)`
	
	`///17.X Implementation`
	`try AWController.clientInstance().canHandle(protectionsSpace:`
	`challenge.protectionsSpace )`

AWLog singleton (Logging)
Use this instead of the AWController to send logs.
	`///5.9.X Implementation`
	`AWLog.sharedInstance().sendApplicationLogs(success, errorName)`
	
	`///17.X Implementation`
	`AWController.clientInstance().sendLogDataWithCompletion { (success, error)`
	`}`

Component	Sample Code

Network status
Update the front of the enum to AWSDK.
	`///5.9.X Implementation`
	`AWNetworkActivityStatus`
	
	`///17.X Implementation`
	`AWSDK.NetworkActivityStatus`

Profiles and profile payloads
Drop the **AW** from the front of profiles.
	`///5.9.X Implementation`
	`AWProfile`
	
	`///17.X Implementation`
	`Profile`

Custom settings
Access custom settings through **AWController** instead of **AWCommanManager**.
	`///5.9.X Implementation`
	`AWCommandManager().sdkProfile().customPayload`
	
	`///17.X Implementation`
	`AWController.clientInstance().sdkProfile()?.customPayload`

Account object
The account object is now a property on **AWController** instead of an accessor method.
This property returns default, empty values for SAML and token enrollment.
	`///5.9.X Implementation`
	`AWController.clientInstance().account()`
	
	`///17.X Implementation`
	`AWController.clientInstance().account`

Component	Sample Code
User credentials	`///5.9.X Implementation` `AWController.clientInstance().updateUserCredentials(completions: {` `error) in {` `...` `} )` `///17.X Implementation` `AWController.clientInstance().updateUserCredentials(with: { (success` `{` `...` `})`
OpenInURL calls	`///5.9.X Implementation` `AWController.clientInstance().handleOpen(url,` `fromApplication: sourceApplication)` `///17.X Implementation` `AWController.clientInstance().handleOpenURL(url,` `fromApplication: sourceApplication)`
DeviceInformationController Replace **MDMInformationController** with **DeviceInformationController**.	NA
Manually load commands Use an API on **AWController** to force commands to reload instead of using the command manager.	`///5.9.X Implementation` `AWCommandHandler.sharedHandler().loadCommands()` `///17.X Implementation` `AWController.clientInstance().loadCommands()`

Install the AirWatch SDK for iOS (Swift) in a New Environment

Install the SDK in an environment without a previous version of the SDK.

For details on how to expose a custom scheme for the call back scheme using the AirWatch Agent for iOS, see Expose a Custom Scheme To Use in a Callback Scheme on page 12.

1. Unzip the **AirWatchSDK DMG** file.
2. Drag and drop the current **AirWatchSDK** framework file and the attached **AWCMWrapper** file into your **Embedded Binaries** step in the build phase section of your project settings.

   Do not add the framework files to only the **Link Binary with Libraries** because this actions causes the application to crash. When you add it to the **Embedded Binaries**, this action automatically adds the file to the **Link Binary with Libraries**, too.
3. Register your callback scheme.
4. Import the **AWSDK** module.
5. Make your **AppDelegate** conform to the **AWControllerDelegate** protocol.

```
        import AWSDK

        class AppDelegate: UIResponder, UIApplicationDelegate,
                 AWSDKDelegate {
```

6. In the **AppDelegate**, add the following code to initialize and start the SDK.

   Do not call the start method in **applicationWillEnterForeground** or **applicationDidBecomeActive**. These start methods result in inconsistent UI behavior.

```
        func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
    [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
       let awcontroller = AWController.clientInstance()
        awcontroller.callbackScheme = "myCallbackScheme"
        awcontroller.start()
        return true
    }
```

7. In the AppDelegate, implement the listed method and code to enable the SDK to receive and handle communication from other AirWatch applications.

```
  func application(_ application: UIApplication, open url: URL, sourceApplication: String?,
annotation: Any) -> Bool {
        // `AWController.handleOpenURL` method will reconnect the SDK back to its previous state
to continue.
        // If you are handling application specific URL schemes. Please make sure that the URL
is not intended for SDK Controller.
        // An example way to perform this.
        let handedBySDKController = AWController.clientInstance().handleOpenURL(url,
fromApplication: sourceApplication)
        if handedBySDKController  {
            AWLogInfo("Handed over open URL to AWController")
            // SDK Controller will continue with the result from Open URL.
            return true
        }

        // Handle if this URL is for the Application.
        return false
    }
```

8. Implement the required delegate method **controllerDidFinishInitialCheck**.

```
         func controllerDidFinishInitialCheck(error: NSError?) {
       if error != nil {
        AWLogError("Initial Check Done Error: \(error)")
        return
     }
     AWLogInfo("SDK Initial Check Done!")
        }
```

You can add optional delegate methods that are described in Required and Optional AWController Delegate Callback Methods on page 16.

Expose a Custom Scheme To Use in a Callback Scheme

You must register a callback scheme to install the AirWatch SDK for iOS (Swift) in an environment with no previous SDK version. Code your application to expose a custom scheme so that it can receive a callback from the AirWatch Agent for iOS.

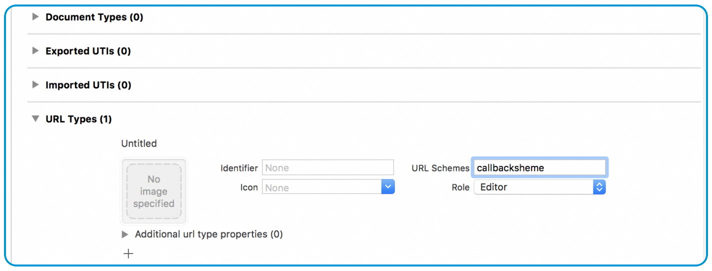
Perform this task in Xcode.

See Install the AirWatch SDK for iOS (Swift) in a New Environment on page 11 for instructions to install the SDK in a clean environment.

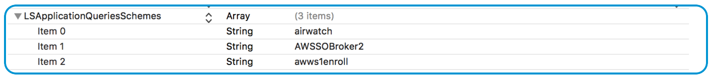
1. In Xcode, navigate to **Supporting Files**.
2. Select the file **<YourAppName>-Info.plist**.
3. Navigate to the **URL Types** section.

   If it does not exist, add it at the **Information Property List** root node of the PLIST.

4. Expand the **Item 0** entry and add an entry for **URL Schemes**.
5. Set the next **Item 0** under **URL Schemes** to the desired callback scheme.



6. Whitelist all AirWatch anchor application schemes under the **LSApplicationQueriesSchemes** entry in the **Information Property List**.



LSApplicationQueriesSchemes	Array	(3 items)
Item 0	String	airwatch
Item 1	String	AWSSOBroker2
Item 2	String	awws1enroll

Chapter 3:
AirWatch SDK for iOS (Swift) Setup

Initialize the AirWatch SDK for iOS (Swift)

Add the listed code to import the SDK and to run the correct protocol. Then start the SDK and setup the callback scheme.

Task	Code
Add the listed code to the AppDelegate.	`import AWSDK`

Code the AppDelegate to use the **AWSDK Delegate** protocol.	`import AWSDK`
	`class AppDelegate: UIResponder, UIApplicationDelegate, AWSDKDelegate {`
	`  ...`
	`}`

Set the AppDelegate, setup the callback scheme, and start the SDK.	`func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool`
	`    {`
	`        // Override point for customization after application launch.`
	`        let awc = AWController.clientInstance()`
	`        awc.delegate = self`
	`        // Your application's scheme name`
	`        awc.callbackScheme = "myCallBackSchemeName"`
	`        awc.start()`
	`        return true`
	`    }`

Task	Code
Set the AppDelegate's class to use the listed protocols.	

```
// Called once the SDK has finished its setup
func initialCheckDone(Error error: NSError?) {
    AWLogDebug("SDK Initial Check Done!")
}

// Called when the configurations profiles have been recieved from console and can now be
accessed from AWController or from the parameter in this call back
func receivedProfiles(profiles: NSArray) {
        AWLogDebug("SDK received Profiles!")
}

// Called when the SDK has locked
func lock() {
        AWLogDebug("SDK locked!")
}

// Called when the SDK has unlocked
func unlock() {
        AWLogDebug("SDK unlocked!")
}

// Called when the SDK has wiped all of its data; the application should wipe any of its
application specific data
func wipe() {
    AWLogDebug("SDK started wiping application!")
}

// Called to alert the application to stop its network activity
func stopNetworkActivity(networkActivityStatus: AWNetworkActivityStatus) {

}

// Called to alert the application to resume its network activity
func resumeNetworkActivity() {

}
```

Required and Optional AWController Delegate Callback Methods

Add the required initial-check method and use optional delegate callback methods that are part of the AWController.

Add these optional methods after you install the SDK. See Install the AirWatch SDK for iOS (Swift) in a New Environment on page 11 for details.

Delegate method	Description
Required methods	
initialCheckDone(Error error: NSError?)	Called once the SDK finishes its setup.
Optional methods	
receivedProfiles(_ profiles: NSArray)	Called when the configurations profiles are received from the AirWatch Console. AWController or the parameter in this callback can now access the configurations profiles.
wipe()	Called when the SDK has wiped all of its data. The application wipes any of its application specific data.
lock()	Called when the SDK has locked, user will need to unlock with username/password, passcode, touch-id in order to access application.
unlock()	Called when the SDK has been unlocked by some form of acceptable authentication (username/password, passcode, touch-id).
stopNetworkActivity(_ networkActivityStatus: NetworkActivityStatus)	Called to alert the application to stop its network activity due to some restriction set by the admin's policies such as cellular data connection disabled while roaming, if airplane mode is switched on, SSID does not match what is on console, proxy failed, etc.
resumeNetworkActivity()	Called to alert the application to resume its network activity because it is now fine to do so based on the device's current connectivity status and policies set by administrator.
userChanged()	Called when the currently logged in user has changed to alert the application of the change.
didReceiveEnrollmentStatus (_ enrollmentStatus: EnrollmentStatus)	Called when the SDK has received the enrollment status of this device from console. The application can now query the SDK for the enrollment status using the **DeviceInformationController** class after this point or use the **EnrollmentStatus** parameter given in this delegate call.

Keychain Access Group Entitlements

Sign the application with the listed component to share data in a keychain access group and to use the AirWatch SDK features.

Enable or Disable Keychain Sharing

Enable keychain sharing entitlements to sign applications with a keychain access group.

Disable keychain sharing to not share data and to sign the application with another string.

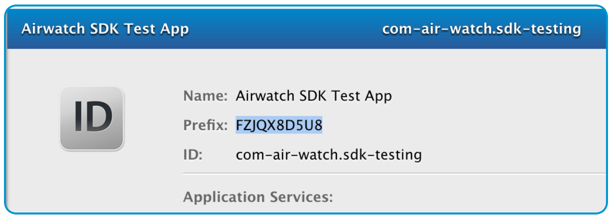
Format of Entitlements

The format for keychain access group entitlements are **\accessGroupName**. The group names are defined in a list and multiple applications have the same **AppIdentifierPrefix** to share date.

The **AppIdentifierPrefix** string associates to the bundle ID of the application. For an application shares data, the applications in the group must share the same keychain access group. You create the bundle ID in the Apple Developer portal and you associate the bundle ID with a prefix or group.

Example

An application is signed with a specific string to share data when you enable or disable keychain sharing.



Airwatch SDK Test App com-air-watch.sdk-testing

Name: Airwatch SDK Test App
Prefix: FZJQX8D5U8
ID: com-air-watch.sdk-testing

Application Services:

- App name - AirWatchSDKTestApp
- AppIdentifierPrefix - FZJQX8D5U8
- BundleID - com.MyCompany.AirWatchSDKTestApp

Keychain sharing enabled	Application signed with the listed string
Yes	With group names as AirWatchSDKTestAppAccessGroup1 and AirWatchSDKTestAppAccessGroup2, the system signs the application with the prefix string. • FZJQX8D5U8.AirWatchSDKTestAppGroup1 • FZJQX8D5U8.AirWatchSDKTestAppGroup2
No	The system signs the application with the bundle ID. • FZJQX8D5U8.com.MyCompany.AirWatchSDKTestApp

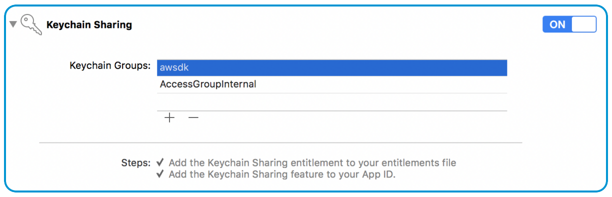
For more information on the SDK and keychain enablement, see Troubleshoot Keychain Enablement on page 19. For the procedure to enable keychain sharing for applications with the same prefix and keychain access groups, see Enable Keychain Sharing for SDK Applications on page 18.

Enable Keychain Sharing for SDK Applications

Enable keychain sharing for SDK-built applications that already share the same AppIdentifierPrefix and the same keychain access group. Perform this task in your Xcode project.

1. In Xcode, select your application’s target and go to **Capabilities**.
2. Go to **Keychain Sharing** and turn it on.
3. Select the plus icon (+) and name the group as **awsdk**.

4. Drag the new access group to the top of the list.



Keychain Groups: awsdk
AccessGroupInternal

Steps: ✓ Add the Keychain Sharing entitlement to your entitlements file
✓ Add the Keychain Sharing feature to your App ID.

Troubleshoot Keychain Enablement

You must enable keychain sharing to use AirWatch SDK for iOS (Swift) features. View some common issues with setting keychain sharing and their solutions.

Disabled Keychain Sharing

Symptom

The SDK cannot initialize because the keychain-saves cannot happen.

Fix

Enable keychain sharing by signing the application with the keychain access group.

Different AppIdentifierPrefix

Symptom

Applications in a keychain access group cannot share passcodes or data if they have different prefixes. The system treats the different prefixes as separate clusters.

Fix

Edit the prefixes for applicable applications on the Apple Developer portal. However, before you change prefixes, ensure you do not need the data stored with the older prefix. This older data is lost when the prefix changes.

Different Keychain Access Groups

Symptom

Applications with the same prefix cannot share passcodes or data if they are in different keychain access groups. The system treats the different groups as separate clusters.

Fix

Ensure that the applicable keychain access groups have enabled keychain sharing.

Merging applications from different groups that use the same account and service names can result in data collisions. Check for the listed situations to prevent collisions.

- The **kSecAttrAccessGroup** attribute is one of the required attribute that can uniquely identify the item stored or retrieved from the keychain.

- All other attributes, for example **kSecAttrAccount** and **kSecAttrService**, that uniquely identify the item stored and retrieved are the same.
- The **kSecAttrAccessGroup** attribute is not specified in the actual query to store and retrieve from the keychain.

More Information

For information on how to sign the application for keychain sharing, see Keychain Access Group Entitlements on page 17. For the procedure to enable keychain sharing for applications with the same prefix and keychain access groups, see Enable Keychain Sharing for SDK Applications on page 18.

See Apple documentation for more information on entitlements and keychains at the listed sites.

- https://developer.apple.com/library/content/technotes/tn2415/_index.html
- https://developer.apple.com/library/content/documentation/Security/Conceptual/keychainServConcepts/02concepts/concepts.html

Cluster Session Management and Reduced Flip Behavior for SSO in the AirWatch SDK for iOS (Swift)

An application built with Swift that uses the SDK does only flips to retrieve account information. It does not flip to the anchor application to retrieve data, like environment information, and to lock and unlock operations.

In the Objective-C SDK, applications needed to flip to the anchor application to retrieve environment information, account details, and to perform all lock and unlock operations.

Cluster Session Management Explanation

The latest SDK for iOS (Swift) introduces a new mechanism using the shared keychain for SDK apps to communicate with other SDK apps on the device. This approach provides benefits from both a security and a user experience perspective.

SDK applications built by the same developer account and that are also in the same keychain group or “cluster” can now share an app passcode and an SSO session without requiring a flip to the Agent, Container, or Workspace ONE every time authentication is required.

However, applications on the same device built by different keychain groups cannot take advantage of this passcode sharing capability. There are some scenarios that still require a flip to the Agent or anchor app to obtain the server URL and other setup information. This particular flip should only occur once per cluster of applications.

Entries to Set in the AWSDKDefaultSettings.plist

Use entries in the AWSDKDefaultSettings.plist to customize the application with AirWatch SDK for iOS (Swift) features. Many of these entries require you to configure their counterparts in the SDK default settings and policies section of the AirWatch Console.

Create the AWSDKDefaultSettings.plist

1. Create a bundle named **AWSDKDefaults**.
2. Create a PLIST named **AWSDKDefaultSettings.plist** and put it in the **AWSDKDefaults** bundle.

Branding, Available Entries

Use the available entries, with the following structure, to add functionality to the application.

- Root (Dictionary)
  - Branding (Dictionary)
    - Colors (Dictionary)
      - EnableBranding (Boolean = YES)
      - PrimaryHighlight (Dictionary)
        - Red (Number = 238)
        - Green (Number = 139)
        - Blue (Number = 48)
        - Alpha (Number = 255)
    - AppLogo_1x (String = logoFileName)
    - AppLogo_2x (String = logoFileName)
    - SplashLogo_1x (String = splashLogoFileName)
    - SplashLogo_2x (String = splashLogoFileName)

QR Scan

Include **NSCameraUsageDescription** in the application info.plist file to enable the SDK to scan QR codes with the device camera.

Provide a description that devices prompt users to allow the application to enable this feature.

FaceID

Include **NSFaceIDUsageDescription** in the application info.plist file to enable the SDK to use FaceID.

Provide a description that devices prompt users to allow the application to enable this feature. Consider controlling the message users read. If you do not include a description, the iOS system prompts users with native messages that might not align with the capabilities of the application.

Test the Integration and Functions of Applications

It is important to test the integration of your application with the AirWatch SDK, including the delivery of profiles from the AirWatch Console to your application.

Initialize the SDK in your application to set communication with the AirWatch server and test the application.

1. Enroll your test device.

   Enroll devices to the AirWatch Console to enable communication between them.

   The SDK does not currently support testing in a simulator.

2. Upload the SDK-built app or a placeholder application that has the same bundle ID as the testing application.

   Create an empty application with the bundle ID of the testing-application to identify the application. Upload the empty application to the console and assign a default or custom SDK profile to it.

3. Assign an SDK profile to the application.

   If you do not assign a profile, the SDK does not initialize correctly.

   This step enables the console to send commands to the application with the record.

4. Push the application to test devices.

   You save the application and assign it using the flexible deployment feature. Flexible deployment rules push the application to test devices with the app catalog. Use devices for testing that are AirWatch managed devices.

   You do not have to repush the application every time you make a change.

5. Run your application in Xcode.

   Run your application in Xcode. The console pushes the initialization data to the application when the application installs on test devices. After the application initializes, you can run the application as many times as you want to debug it.

Chapter 4: AirWatch SDK for iOS (Swift) Capabilities

SDK Capabilities Quick Reference, Code and Console

View if an SDK capability needs both code and console settings, or just one of the two.

SDK Capability	Add Code (Beyond AWController)	Set in the Console
Authentication	Yes Use SDK helper classes.	Yes • Enable • Set a type
SSO	Yes Enable keychain sharing.	Yes Enable
Integrated authentication	Yes Use the challenge handler.	Yes • Enable • Enter allowed sites • Set an authentication option
App tunnel	**No**	Yes • Enable • Select a mode ○ Configure the proxy components of the VMware Tunnel. ○ If not using VMware Tunnel, ensure the integration of the selected proxy with your AirWatch deployment.
Data loss prevention (DLP)	Yes Set the AWSDKDefault bundle and the AWSDKDefaultSettings.plist.	Yes • Enable • Set enable copy and paste • Set enable composing email
Analytics	Yes • Set the AWDataSampler. • Set the AnalyticsHelper. • Decide to use the SDK or the AirWatch Agent for telecom data.	Yes • Enable • Set privacy if setting do not disturb

SDK Capability	Add Code (Beyond AWController)	Set in the Console
Branding	Yes Add values to the AWSDKDefaultSettings.plist.	Yes • Enable • Set colors • Upload images
Sample data and MDM information	Yes • Use the beacon. The SDK automatically sends the beacon but you can manually send the beacon when desired. • Query the DeviceInformationController singleton class.	**No**
Compromised protection	**No** Use code to check the status of devices with the application.	Yes Enable
Custom settings	Yes Use the AWCustomPayload object.	Yes • Enable • Enter code
Geofencing	**No** Devices must use location services and have GPS.	Yes • Enable • Set the area
Logging	Yes Add APIs for logging. See the sample applications for examples.	Yes • Enable • Set the level • Set wi-fi
Offline access	**No**	Yes • Enable • Set time allowed to be offline
Encryption	Yes Use methods in the AWController to encrypt and decrypt data.	**No** However, the strength of the encryption is dependent on the authentication method set in the AirWatch Console.

Application Configuration

Application configuration consists of enabling or disabling features that pertain to iOS or the Airwatch SDK. You must add a bundle and PLIST to allow configuration.

To usethis feature, modify a value in a PLIST file and configure the feature in the console. The SDK handles all the logic for capabilities like data loss prevention (DLP), branding, and swizzling calls.

Set Up the Bundle and the PLIST

1. Create a bundle named **AWSDKDefaults**.

   If iOS does not offer a non-unit testing bundle, add a macOS bundle and modify its build setting as an iOS compatible. To do this, modify the **BaseSDK** to iOS.

2. Add bundle to the **Bundle Resources** of your application.

3. Create a PLIST named A**WSDKDefaultSettings.plist** and place it into the **AWSDKDefaults** bundle.

Authentication Type Function Description

Set access to your application with the authentication type function. Use a local passcode, AirWatch credentials, or require no authentication.

Select an authentication type in the AirWatch Console and use the provided SDK helper classes in your application.

Setting	Description
Passcode	Designates a local passcode requirement for the application. Device users set their passcode on devices at the application level when they first access the application.
Username and Password	Requires users to authenticate to he t application with their AirWatch credentials.
Disabled	Requires no authentication to access the application.

Authentication Type and SSO Setting Behaviors

You can use keychain sharing, the authentication type, and the single sign on (SSO) option to make access to your application persistant.

Keychain Access Group Required

You must have a shared space, a keychain access group, so that applications signed in the correct format can share keychain entries. See Keychain Access Group Entitlements on page 17 for information on the signing format. See Troubleshoot Keychain Enablement on page 19 for common issues with keychain sharing.

Enable Authentication Type and SSO

If you enable both authentication type and SSO, then users enter either their passcode or credentials once. They do not have to reenter them until the SSO session ends.

Enable Authentication Type Without SSO

If you enable an authentication type without SSO, then users must enter a separate passcode or credentials for each individual application.

Authentication and Changes to Active Directory Passwords

Use an API to update the AirWatch SDK for iOS (Swift) credentials when the credentials change.

If an Active Directory (AD) password changes and becomes out of sync with the object account of the SDK, use an API to update the SDK credentials.

```
/// Swift
AWController.clientInstance().updateUserCredentials(with: { (success, error) in {
        ///insert completion handler code here
}
```

Find the new credentials in the SDK account object after the callback successfully returns.

SSO Configurations and System Login Behavior

AirWatch allows access to iOS applications with single sign on enabled in two phases. AirWatch checks the identity of the application user and then it secures access to the application.

Requirements for Use in Applications that Use SDK Functions

To use the SSO function, ensure these components are set.

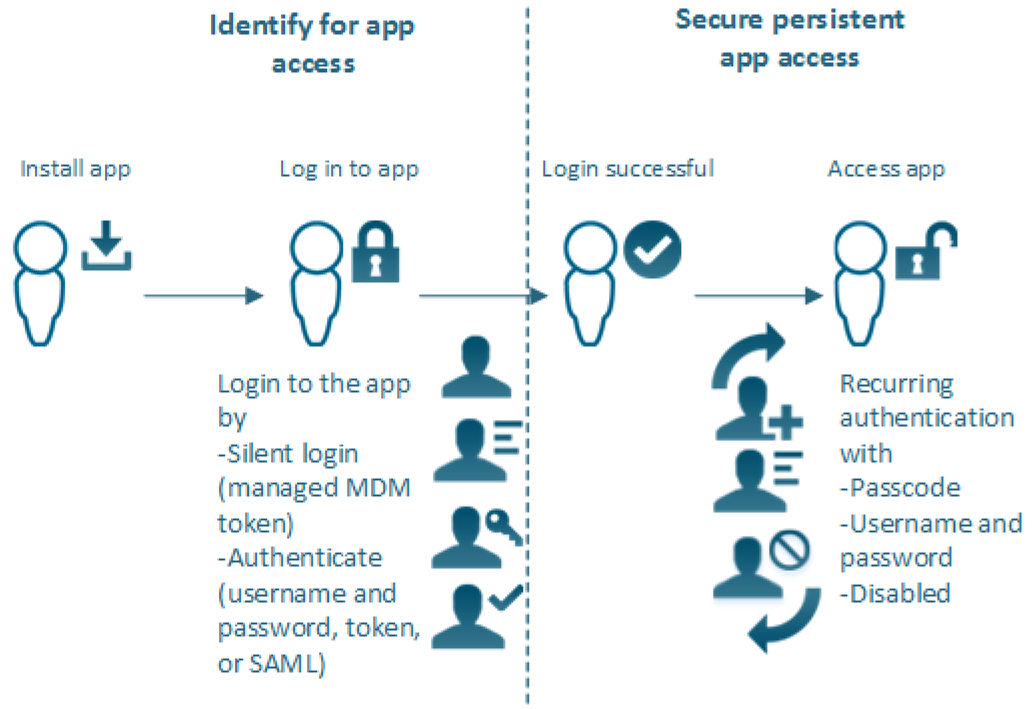
- Enable the SSO setting in the SDK default settings and policies in the AirWatch Console.
- Initialize the SDK in the AppDelegate.
- Ensure an anchor application is on devices like the AirWatch Agent or Workspace ONE. The anchor application deployment is part of the AirWatch mobile device management system.

Query the Current SSO Status

To query the SSO status of the iOS application, wait for the **initialCheckDone** method to finish. Look in the **DeviceInformationController** class for the **ssoStatus** property. If the **initialCheckDone** method is not finished, the SSO status returns as **SSO disabled**.

Application Access With SSO Enabled

The authentication process to an application with AirWatch SSO enabled follows the general depiction.


Identify for app access
Secure persistent app access
Install app
Log in to app
Login successful
Access app
Login to the app by
-Silent login (managed MDM token)
-Authenticate (username and password, token, or SAML)
Recurring authentication with
-Passcode
-Username and password
-Disabled

The first phase ensures that the user's credentials are valid. The system identifies the user first by silent login. If the silent login process fails, then the system uses a configured, authentication system. AirWatch supports username and password, token, and SAML.

The second phase grants the user access to the application and keeps the session live with a recurring authentication process. AirWatch supports passcode, username and password, and no authentication (disabled).

Authentication Behavior By SSO Configuration

The SSO configuration controls the login behavior users experience when they access applications. The authentication setting and the SSO setting affect the experience of accessing the application.

Authentication phase	SSO enabled	SSO disabled
Passcode		
Identify	**Silent login**: The system registers credentials with the managed token for MDM. If silent login fails, the system moves to the next identification process. **Authenticate**: The system identifies credentials against a common authentication system (username and password, token, and SAML).	**Silent login**: The system registers credentials with the managed token for MDM. If silent login fails, the system moves to the next identification process. **Authenticate**: The system identifies credentials against a common authentication system (username and password, token, and SAML).

Authentication phase	SSO enabled	SSO disabled
Secure	**Prompt if passcode exists**: The system does not prompt for the passcode if the session instance is live. **Prompt if passcode does not exist**: The system prompts users to create a passcode. **Session shared**: The system shares the session instance across applications configured with AirWatch SSO enabled.	**Prompt if passcode exists**: The system prompts users the application passcodes. **Prompt if passcode does not exist**: The system prompts users to create a passcode. **Session not shared**: The system does not share the session or the passcode with other applications.
Username and password		
Identify	**Silent login**: The system registers credentials with the managed token for MDM. If silent login fails, the system moves to the next identification process. **Authenticate**: The system identifies credentials against a common authentication system (username and password, token, and SAML).	**Silent login**: The system registers credentials with the managed token for MDM. If silent login fails, the system moves to the next identification process. **Authenticate**: The system prompts for application login credentials.
Secure	**Prompt**: The system does not prompt for the login credentials if the session instance is live. **Session shared**: The system shares the session instance across applications configured with AirWatch SSO enabled.	**Prompt**: The system prompts for the login credentials for the application on every access attempt. **Session not shared**: The system does not share the session with other applications.
Disabled		
Identify	**Silent login**: The system registers credentials with the managed token for MDM. If silent login fails, the system moves to the next identification process. **Authenticate**: The system identifies credentials against a common authentication system (username and password, token, and SAML).	**Silent login**: The system registers credentials with the managed token for MDM. If silent login fails, the system moves to the next identification process. **Authenticate**: The system prompts for application login credentials.
Secure	**Prompt**: The system does not prompt users for authentication.	**Prompt**: The system does not prompt users for authentication.

Integrated Authentication and the Challenge Handler

Use integrated authentication to pass single sign on (SSO) credentials or certificates to use to authenticate to web sites like content repositories and wikis. Set the function in the AirWatch Console and add a list of allowed sites. Then use the challenge handler in your application to handle incoming authentication challenges.

See Configure Integrated Authentication for the Default SDK Profile for information on setting integrated authentication in the AirWatch Console.

Challenge Handler Methods for Challenges

Find the challenge handler in the AWController class of the SDK. Inside the AWController, use the listed methods to handle an incoming authentication challenge for connections made with NSURLConnection and NSURLSession.

Method	Description
func canHandle(_ protectionSpace: URLProtectionSpace, withError error: Error?) -> Bool	Checks that the AirWatch SDK can handle this type of authentication challenge. The SDK makes several checks to determine that it can handle challenges. 1. Is the Web site challenging for authentication on the list of allowed sites in the SDK profile? 2. Is the challenge one of the supported types? • Basic • NTLM • Client certificate 3. Does the SDK have a set of credentials to respond? • Certificate • User name and password If all three of the criteria are met, then this method returns **YES**. The SDK does not handle server trust, so your application must handle **NSURLAuthenticationMethodServerTrust**.
func handleChallenge(forURLSessionChallenge challenge: URLAuthenticationChallenge, completionHandler: @escaping (_ disposition: URLSession.AuthChallengeDisposition, _ credential: URLCredential) -> Void) -> Bool	Responds to the actual authentication challenge from a network call made using **NSURLSession**. This method is the same as the handleChallenge method, except the system uses this method with calls made with NSURLSession. This call involves using a completion block to handle authentication challenges.

Requirements for Integrated Authentication

Ensure to set the listed configurations so that integrated authentication works.

- The URL of the requested web site must match an entry in your list of **Allowed Sites**.
- The system must make the network call so that the process provides an **NSURLAuthenticationChallenge** object.
- The web site must return a 401 status code that requests authentication with one of the listed authentication methods.

    - NSURLAuthenticationMethodBasic
    - NSURLAuthenticationMethodNTLM
    - NSURLAuthenticationMethodClientCertificate

- The challenge handler can only use the enrollment credentials of the user when attempting to authenticate with a web site. If a web site requires a domain to log in, for example ACME\jdoe, and users enrolled with a basic user name, like jdoe, then the authentication fails.
- For applications using WebView, use SDK's **handleChallenge** method in the URLSession's challenge handler. Display the response on a UIWebView or a WKWebView. Do not use the SDK's handleChallenge method directly inside WKWebView's challenge handler.

VMware Tunnel for App Tunneling by Proxy Components

The proxy components of the VMware Tunnel provides a secure method for individual applications that use the AirWatch SDK to access corporate resources.

The SDK for iOS (Swift) provides app tunneling without adding code to the application. However, you need to configure app tunneling in the AirWatch Console.

Configure App Tunneling and Split Tunneling

To configure app tunneling in the console, use the AirWatch App Tunnel settings.

1. Navigate to **Groups & Settings > All Settings > Settings & Policies > Security Policies > AirWatch App Tunnel**.
2. **Enable** the setting.
3. Select an app tunnel mode.

   Select **VMware Tunnel - Proxy** if your company has this configured.
4. In the **App Tunnel URLs** field, enter the URLs that you do not want to tunnel.
   - Enter no URLs and every URL goes through the VMware Tunnel.
   - Enter one or more URLs and the system splits the traffic. This configures split tunneling. The system does not send the URLs entered in this field through the VMware Tunnel. The system does send all other URLs through t he VMware Tunnel.

VMware Tunnel Proxy Documentation

The Tunnel proxy component uses HTTPS tunneling to use a single port to filter traffic through an encrypted HTTPS tunnel for connecting to internal sites such as SharePoint or a wiki.

For more information about Tunnel proxy components, see the AirWatch Console Online Help topic Proxy (SDK/Browser) Architecture and Security.

Use DLP to Prevent Copy and Paste

The data loss prevention (DLP) function to prevent copy and paste actions requires a setting in the AirWatch Console and added values to the AWSDKDefaultSettings.plist. After you set the console and add the PLIST values, the SDK handles all logic for DLP.

Configure the AirWatch Console

1. Set the **Enable Copy and Paste** function to **No** in the console.
2. Add a new boolean entry entitled **AWClipboardEnabled** and set it to **Yes** to enable prevention of copy and paste.

For information on setting DLP, see the VMware AirWatch Online Help topic Configure Data Loss Prevention for the Default SDK Profile.

Add Values to AWSDKDefaultSettings.plist

Inside your **AWSDKDefaultSettings.plist** add a new boolean entry entitled **AWClipboardEnabled** and set it to **YES** to enable prevention of copy and paste.

See Entries to Set in the AWSDKDefaultSettings.plist on page 20 for information on creating the AWSDKDefaultSettings.plist.

Use DLP to Control Links to Open in VMware Browser, AirWatch Boxer, or AirWatch Inbox

Configure applications built with the AirWatch SDK to open in the VMware Browser and to compose emails in AirWatch Boxer or AirWatch Inbox. This feature enables end users to use alternative systems other than Safari and the Mail app. To develop this feature, create a bundle in your iOS application and configure AirWatch to enforce the behaviors in the bundle.

Configure both systems, the browser and email systems, for this feature to work. Perform the procedures in the listed order.

1. Initial Set Up of the Bundle and PLIST
2. Enable Links for Browser
3. Enable Links for Inbox
4. Contain Data to Browser and Inbox

Initial Set Up of the Bundle and PLIST

Perform these steps before you enable any links. Use this bundle and PLIST for both HTTP/HTTPS links and MAILTO links.

1. Create a bundle named `AWSDKDefaults`.
2. Create a PLIST named `AWSDKDefaultSettings.plist` and put it in the `AWSDKDefaults` bundle.

Enable Links for Browser

To enable the application to open HTTP / HTTPS links in the VMware Browser, enable a few dictionary and PLIST flags.

1. Work in the `AWSDKDefaults` bundle.
2. Create a dictionary named `AWURLSchemeConfiguration` and put it in the `AWSDKDefaultSettings.plist`.
3. Inside the `AWURLSchemeConfiguration` dictionary, create a new Boolean entry with the key name **enabled** and set the Boolean value to **Yes**.

   If you set the Boolean value to **No**, then the HTTP and HTTPS links open in Safari. If set to **Yes**, then your SDK app opens in VMware Browser.

Enable Links for Boxer or Inbox

To enable the application to open MAILTO links in AirWatch Boxer or AirWatch Inbox, enable a few dictionary and PLIST flags.

1. Work in the `AWSDKDefaults` bundle.
2. Create a dictionary named `AWMaitoSchemeConfiguration` and put it in the `AWSDKDefaultSettings.plist`.
3. Configure the `AWMailtoSchemeConfiguration` dictionary, create a new Boolean entry with the key name as **enabled** and set the Boolean value to **Yes**.

   If you set the Boolean value as **No**, then MAILTO links open in the native mail. If set to **Yes**, then your SDK app looks to see if you enabled data loss prevention in the SDK profile.

   - DLP Enabled – The app opens in AirWatch Boxer or AirWatch Inbox.
   - DLP Disabled – The app opens in the iOS Mail app.

Contain Data to Browser and Inbox

Use the data loss prevention, DLP, settings in the AirWatch default SDK profile to enforce the application to use VMware Browser and AirWatch Boxer or AirWatch Inbox.

If you do not enable data loss prevention in the SDK policy, the application opens links in Safari and composes email in the iOS Mail app.

1. Navigate to **Groups & Settings > All Settings > Apps > Settings and Policies > Security Policies**.
2. Select **Enabled** for **Data Loss Prevention**.
3. Disable the **Enable Composing Email** check box for the MAILTO links. If you do not disable this option, the application opens from the Mail app and not from Inbox.

Limitation With MFMailComposeViewController

If you use the `MFMailComposeViewController` scheme in your MessageUI framework, this functionality is not supported. The system cannot specify how end users access your application when it is an attachment in an email. End-users access the application with the Mail app and not Inbox.

SupportInformationController

The **SupportInformationController** class allows you to query for the email address and telephone numbers for contacting enrollment support which you can display on the application UI.

Set Up the DataSampler Module for Analytics

The DataSampler module samples detailed device data and reports it back to the AirWatch Console. Device details such as analytics, call logs, GPS location, and network adapters are all sampled with the DataSampler.

Important: For GPS sampling to function, ensure your application supports location tracking. For more information, see Apple's documentation at https://developer.apple.com/documentation/corelocation.

The DataSampler samples and transmits on two different time intervals. Device samples remain on to the disk and the system removes them after transmitted. This process allows the developer to sample statistics multiple times before sending them to AirWatch. Samples stored on the disk are useful when a device does not have network connectivity.

AWDataSampler is a singleton object. There can only be one DataSampler for each process.

Configuration

These parameters are required to set up a DataSampler.

- **sampleModules** – Names the bitmask whose flags specify which modules to use.
- **defaultSampleInterval** – Specifies the time in seconds between DataSampler samples for all modules by default.
- **defaultTransmitInterval** – Specifies the time in seconds between DataSampler transmissions for all modules by default.
- **traceLevel** – Determines the error and information logging level of the DataSampler module when it is running.

Modules Available for Sampling

These modules are available for sampling in the DataSampler.

- **AWDataSamplerModuleSystem**
- **AWDataSamplerModuleAnalytics**
- **AWDataSamplerModuleGPS**
- **AWDataSamplerModuleNetworkData**
- **AWDataSamplerModuleNetworkAdapter**
- **AWDataSamplerModuleWLAN2Sample**

Gather Telecom Data

Disable the **AWDataSamplerModuleNetworkData** mask if you gather telecom data using the AirWatch Agent. If you enable this mask for the SDK, then you receive duplicate data from the Agent and from the SDK.

Set Do Not Disturb

You can use the SDK to set the do-not-disturb (DND) status on the AirWatch server. You must enable the DND policy in the AirWatch Console. You can find the policy at **Groups & Settings > All Settings > Devices & Users > General > Privacy > DO NOT DISTURB section**.

The two relevant methods are **fetchDeviceDNDStatus** and **setDeviceDNDStatus** found in the **AWDeviceDNDStatus** object.

AnalyticsHelper

The **AnalyticsHelper** is a singleton with a property and a function. Send your custom analytics event from your application to the console with this process.

1. Ask your admin to enable the Analytics setting in the SDK profile for the SDK-built application. This setting is in the console at **Groups & Settings > All Settings > Apps > Settings and Policies > Settings > Analytics**.
2. In the application, call the **recordEvent** method on the singleton after the **controllerDidFinishInitialCheck** delegate callback returns.

```
func sendAnalytics() {
       let analytics = AnalyticsHandler.sharedInstance
       analytics.recordEvent(AWSDK.AnalyticsEvent.customEvent, eventName: "EVENT_NAME",
eventValue: "EVENT_VALUE", valueType: AWSDK.AnalyticsEventValueType.string)
   }
```

After the system records the event, it saves the event in the SDK container for two hours. After the two hours passes, the SDK sends analytics recorded to disk to the console the application re-starts.

Locate the data in the console **in Apps & Books > Applications > Logging > SDK Analytics**.

Use Branding to Add Logos and Primary Highlight Colors

Use the branding function to add logos and primary highlights to your application to customize the look of the application.

Branding by Organization Group

Many organizations brand applications according to the applications assigned organization group in the AirWatch Console. This technique is useful for updating the branding elements inline for time-sensitive events or marketing initiatives.

Access Branding Settings in the SDK

The branding profile is downloaded and available once the **controllerDidReceive(profiles: [Profile])** function is called. Within the branding profile it is possible to view the raw values set in the console.

```
let brandingPayload = AWController.clientInstance().sdkProfile()?.BrandingPayload
```

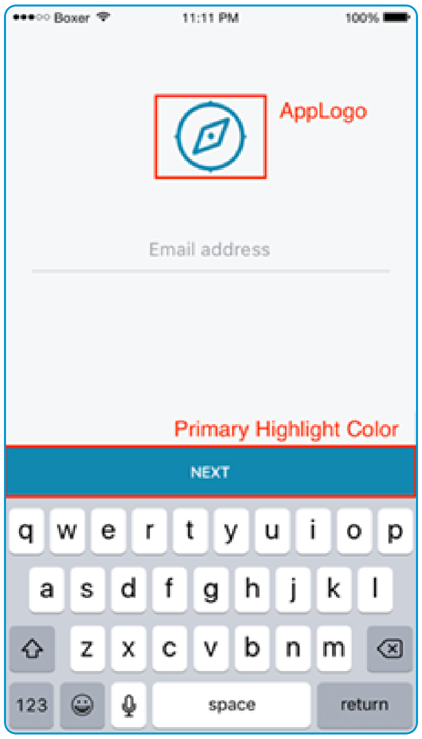
The values in **AWBranding** become set once the **controllerDidFinishInitialCheck**. If a value is not set in the console, then the system returns nil.

Add Values to AWSDKDefaultSettings.plist

You can add a primary highlight color to brand the buttons on the authenication screen. You can also add two company logos (**AppLogo** and **SplashLogo**) within the **Branding** dictionary inside your **AWSDKDefaultSettings.plist**.

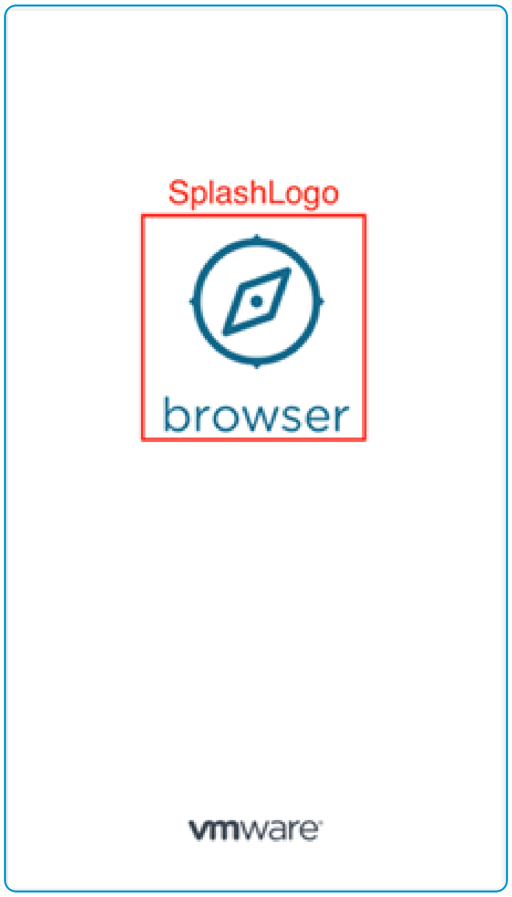
AppLogo

The SDK puts the **AppLogo** on all of the authentication screens.



SplashLogo

The SDK puts the **SplashLogo** on the loading screen and on the second application login screen.



Available Branding Entries in the AWSDKDefaultSettings.plist

Entry	Type
Branding	Dictionary
Colors	Dictionary
PrimaryHighlight	String
AppLogo_1x	String
AppLogo_2x	String
SplashLogo_1x	String
SplashLogo_2x	String

Beacon Data Sent Upon Application Unlock or Sent Manually

The beacon is a regular update sent from the AirWatch SDK for iOS to the AirWatch Console. The SDK sends this data every time it is unlocked. You can also force the beacon when you want data.

The beacon update contains the listed information.

Type of Information	Data
General	• Device name • Organizational group • Application bundle identifier
Platform	• Device operating system (Apple, iOS) • Device operating system version
User	• User email • User full name • User display name
Enrollment	• Device enrolled • Device unenrolled • Device wipe pending
Compliance	• Device compliance • Application compliance

Send the Beacon Manually

Use an API to send the beacon manually.

```
let beaconTransmitter = SDKBeaconTransmitter.sharedTransmitter()

//To send immediately
beaconTransmitter.sendDeviceStatusBeacon(completion: SendBeaconCompletion?)
beaconTransmitter.sendBeacon(updatedAPNSToken: String, completion: SendBeaconCompletion?)

//To start a schedule of how frequenlty to send (If given time interval is less than 60, frequency
will default to 60)
public func startSendingDeviceStatusBeacon(transmitFrequency: TimeInterval = 60)

//To stop the sending the scheduled beacon
public func stopSendingDeviceStatusBeacon()
```

Certificate Pinning

Use certificate pinning to help prevent man-in-the-middle (MITM) attacks by enabling an additional layer of trust between listed hosts and devices.

Certificate pinning requires no code. In the AirWatch Console, enable SSL pinning and upload your certificate.

For information on SSL Pinning, see the AirWatch Online Help topic System / Security / SSL Pinning.

For information on certificates, see the AirWatch Online Help topic Supported Certificate Authorities.

Check the Compromised Status of Devices with Compromised Protection

AirWatch detects jailbroken devices and can wipe compromised devices if enabled in the AirWatch Console.

Compromised protection requires no code unless you want to manually check the status of the device.

Check Compromised Protection Status

To check the status of a device directly in your application, whether the device is online or offline, call the **isCurrentDeviceCompromised()** API from the **DeviceInformationController** singleton class.

```
//Swift
let deviceInfoController = DeviceInformationController.sharedController()
let compromisedStatus = deviceInfoController.isCurrentDeviceCompromised()
if compromisedStatus == true {
  AWLogDebug("My device is jailbroken!")
}
```

Compliance and Compromised Protection

Compromised protection is a feature that allows the developer to check the SSO status and the jailbreak status.

Manually Checking for the Compromised Status of the Device

You can check the compromised status of the device directly in your application by calling the **isCurrentDeviceCompromised()** API from the **DeviceInformationController** singleton.

```
let deviceInfoController = DeviceInformationController.sharedController()
let compromisedStatus = deviceInfoController.isCurrentDeviceCompromised()
if compromisedStatus == true {
  AWLogInfo("My device is jailbroken! Oh no!")
}
```

Custom Settings for the SDK

The SDK allows you to define your own custom settings for your application using an SDK profile.

You can paste raw text in the custom settings section, and the SDK makes this content available inside the application using the **AWCustomPayload** object.

You can define an XML, JSON, key-value pairs, CSV, or plain text for your settings. Parse the raw text in the application once it is received.

Query Devices for MDM Information with DeviceInformationController

Use the **DeviceInformationController** singleton class to query devices for mobile device management (MDM) information.

The class returns the listed MDM information.

- Enrollment status
- Compliance status
- Managed status
- Management type
- Organizational group name
- Organizational group ID
- Device services URL
- Single sign on status
- Compromised status

Requery Method

The method queries the console to send to the containing device a query command to collect certain types of device information.

Geofence Capabilities

A geofence limits the use of devices to specific areas including corporate offices, school buildings, and retail department stores. Geofence settings are configured within the SDK profile and do not require code.

See Geofences for details on how to set Geofences, and see Configure Geofencing for the Default SDK Profile for details on enabling it for the default SDK profile.

The feature works on devices that run location services. To turn on the location services, the device must be connected to either a cellular network, Wi-Fi, or the device must have integrated GPS capabilities. If a device is in "Airplane Mode," the location services are deactivated, and geofencing stops working.

Logging Capabilities

The logging module enables the discovery of bugs or issues when the application is deployed to users.

Set Logging in the Application

Add APIs to call functions and methods for log statements. See the sample application for examples.

Set the Logging Level in the SDK Profile

You must set the logging level in the default profile for the SDK in the AirWatch Console. This configuration ensures that your network is not burdened with unwanted logging activity.

1. Navigate to **Groups & Settings > All Settings > Settings & Policies > Settings > Logging**.
2. Enable the feature.
3. Select the logging level.

Level	Logging Syntax	Description
Error	AWLogError("{log message}")	Records only errors. An error displays failures in processes such as a failure to look up UIDs or an unsupported URL.
Warning	AWLogWarning("{log message}")	Records errors and warnings. A warning displays a possible issue with processes such as bad response codes and invalid token authentications.
Information	AWLogInfo("{log message}")	Records a significant amount of data for informational purposes. An information logging level displays general processes, warning, and error messages.
Debug or Verbose	AWLogVerbose("{log message}")	Records all data to help with troubleshooting. This option is not available for all functions.

4. Set to send logs over wifi or not.

Access SDK and Wrapped App Logs by Log File

Access SDK application logs from the App Logs page.

1. Navigate to **Apps & Books** > **Applications > Analytics > App Logs**.
2. Download or delete logs using the actions menu.

Access Logs by the View Logs Page

Use the View Logs feature from the actions menu to quickly access available log files pertaining to a single application.

1. Navigate to **Apps & Books > Applications > Native** and select the **Internal** tab.
2. Select the application.
3. Select the **View Logs** option from the actions menu.

Offline Access

The offline access function allows access to the application when the device is not communicating with the network. It also allows access to AirWatch applications that use the SSO feature while the device is offline.

Offline Behavior

The SDK automatically parses the SDK profile and honors the offline access policy once **AWController** is started. If you enable offline access and an end-user exceeds the time allowed offline, then the SDK automatically presents a blocker view to prevent access into the application. The system calls the lock method of the **AWSDKDelegate** so your application can act locally.

Encrypt Data on Devices

The SDK for iOS (Swift) offers the use of basic encrypt and decrypt methods to operate on raw data that the system encrypts using the SDK's internal encryption keys.

These methods are defined in the **AWController**.

Important: Do not use these encryption methods on any mission critical data or data that you cannot recover. Examples of unrecoverable data include no backup on a server or if the data cannot be re-derived through other means. The encrypted key (and associated encrypted data) is lost in the event that an end user deletes the application or if an enterprise wipe.

Prequisites

Before you call the encryption methods, ensure the **AWControllerDelegate** receives no errors.

Swift

Applications must ensure that **AWControllerDelegate** receives the **controllerDidFinishInitialCheck(error: NSError?)** callback with no errors before they call the encryption methods.

Objective-C

The **AWControllerDelegate** callback method is **- (void)initialCheckDoneWithError:(NSError * _Nullable)error;**

Encryption Strength and Authentication Mode

The strength of the encryption depends on the enabling of the authentication mode.

If you set authentication passcode or username and password, then the system derives the key used for encryption from the passcode or username and passcode the user enters. The system keeps the key in device volatile memory for additional security.

If you disable authentication, the system randomly generates the encryption key and persists it in device storage.

Encrypt Data not Stored with Core Data

The AirWatch SDK for iOS (Swift) provides the ability to encrypt data that Core Data does not store. These methods take in the data input and return back either the encrypted or decrypted data. These methods are only used for the transformation of the data. The application developer is responsible for the storage of the encrypted data.

Encryption Methods

Swift

```
public func encrypt(_ data: Data) throws -> Data
public func decrypt(_ data: Data) throws -> Data
```

Objective-C

```
- (NSData * _Nullable)encrypt:(NSData * _Nonnull)data error:(NSError * _Nullable
* _Nullable)error SWIFT_WARN_UNUSED_RESULT;
```

```
- (NSData * _Nullable)decrypt:(NSData * _Nonnull)data error:(NSError * _Nullable
* _Nullable)error SWIFT_WARN_UNUSED_RESULT;
```

Error Codes Defined

The enum **AWSDKCryptError** defines the error codes for the error thrown by the methods.

Examples

Encrypt

```
let controller = AWController.clientInstance()
let plainData: Data = .. //assign data to be encrypted
do {
    let encryptedData = try controller.encrypt(plainData)
    //save encryptedData for future use
    //...
} catch let error {
    print(" failed to encrypt data with error: \(String(describing: error))")
}
```

Decrypt

```
let controller = AWController.clientInstance()
let encryptedData = ..//fetch data previously encrypted using Encrypt method above

do {
    let decryptedData = try controller.decrypt(encryptedData)
   //do something with decryptedData
   //...
} catch let error {
    print(" failed to encrypt data with error: \(String(describing: error))")
}
```

Chapter 5: AirWatch SDK for iOS (Swift) and the Apple App Review Process

Overview of the Steps to Configure App Review Mode

Deploy apps that use the AirWatch SDK for iOS to the App Store without dependency on other AirWatch components. The SDK includes a mode for your application for use during the Apple App Review process.

This app review mode removes dependencies on the broker applications such as the AirWatch Agent for iOS, AirWatch Container, and Workspace ONE. It also enables the app reviewer to access the application without enrolling with AirWatch.

Important: Use this work flow only on applications built with the AirWatch SDK that you submit to the App Store for review. Do not use this work flow for any other application development processes. Also, do not use the process in a production environment. This process is only supported for use in a test environment for applications you submit to Apple's App Review.

App review mode includes several steps.

1. Integrate the SDK with your application.
2. Configure the app review mode testing environment in the AirWatch Console, upload the application IPA file, assign it an SDK profile, and deploy it to the test environment. See Configure an App Review Mode Testing Environment in the AirWatch Console on page 46.
3. Assign an app review mode server and a group ID to the SDK PLIST. See Declare the App Review Server and Group ID in the SDK PLIST on page 47.
4. Test the IPA in the test environment. See Test the App Review Mode Testing Environment in the AirWatch Console on page 48.
5. Run the app store build script. See Build Script Information for App Store Submission on page 48.
6. Submit your application for review to the Apple App Store ensuring to add the app review mode server, group ID, and user credentials from the test environment to the submission.

Configure an App Review Mode Testing Environment in the AirWatch Console

With help from the AirWatch Admin, configure a testing environment in the AirWatch Console. Upload your application to this environment so that the app reviewer can review your application without dependencies on other AirWatch components.

Prerequisites

- Integrate the AirWatch SDK with your application.
- You need AirWatch system admin permissions to configure these components. If you do not have these permissions, ask your AirWatch Admin for help.
- Ensure that you create a testing environment that hosts no production applications and components. Use this app review mode environment only for the app review process.
- Configure all options in the app review organization group.

1. Create an App Review Organization Group

Configure a special organization group for app review mode in the AirWatch Console. Record the group ID for later entry to the SDK PLIST.

2. Create a User with Credentials for the Apple App Reviewer

Configure an app review mode user with credentials in the AirWatch Console. You give these credentials to the app reviewer so record the credentials.

3. Create a Smart Group and Add the User

AirWatch deploys applications based on assignment groups, specifically the smart group type. Create a smart group and add the user to the group.

4. Configure the SDK Profile

Use the default SDK profile or a custom SDK profile. Whatever SDK profile you use, ensure that the desired SDK features are enabled. Features to review are the Authentication Type, Single Sign On, and the App Tunnel Mode.

5. Upload the Application to the AirWatch Console

Upload the application binary (IPA) to the internal application area or the public application area of the AirWatch Console. It does not matter which type you use. However, ensure that you assign the SDK profile to the application and assign the test smart group to the application.

The bundle identifier must match the application submitted to the App Review process.

6. Disable Required MDM Enrollment

Disable the requirement for MDM enrollment so that the app reviewer can access the application without enrolling with MDM. Follow these steps to disable MDM enrollment in the AirWatch Console.

Although the setting are nested under the Content Locker, it applies to all applications. Improvements to the user interface are planned for the future.

1. Ensure you are in the app review mode organization group.
2. Navigate to **Groups & Settings > All Settings > Content > Applications > Content Locker**.
3. In the **General** area, disable **Require MDM Enrollment**.
4. Select **Save** to complete the procedure.

Declare the App Review Server and Group ID in the SDK PLIST

To prepare to submit your application to the Apple App Review process, add the app review mode server URL and the group ID. These strings allow the reviewer to review your application without the need for other AirWatch components.

1. If you have not done so, in your Xcode project, create a bundle named **AWSDKDefaults**. if you haven't already done so.

2. If the **AWSDKDefaults** bundle does not have a PLIST named **AWSDKDefaultSettings.plist**, create this PLIST in the bundle.
3. Create a key in the PLIST with the data type string. Name this key **com.vmware.air-watch.enrollment.test-server-url**. This name is case sensitive.
4. Set the value of this key to the server URL of the AirWatch environment you setup in Configure an App Review Mode Testing Environment in the AirWatch Console on page 46.

   Ensure to meet these requirements for the URL.

   - Include **https://** before the URL.
   - Ensure the URL is the exact device services server URL. Do not use the console or API server URL.
   - Do not include **/deviceservices** at the end of the URL. The SDK appends this automatically.
5. Create another key in the PLIST with the data type string. Name this key **com.vmware.air-watch.enrollment.test-org-group-id**. This name is also case sensitive.
6. Set the value of this key to the group ID of the app review group you setup in Configure an App Review Mode Testing Environment in the AirWatch Console on page 46.

Test the App Review Mode Testing Environment in the AirWatch Console

Test that the IPA file, server URL, group ID, and user credentials work before you submit the application for review.

1. Attempt to run the app on a device without any previous app data. This action ensures that stale URL and device information is not present on the device. It also ensures there are no other AirWatch apps on the device.
2. Enter the server URL and group ID when the app prompt for these options.
3. Enter the user credentials when prompted.
4. If the SDK permits you to continue without error and **initialCheckDone** is called, the test environment and components are successful.

Build Script Information for App Store Submission

This process requires a separate build script that you run before you submit the application for review.

Reason for the Special Script

Run the build script to strip the simulator architectures. The application fails the Apple App Review static analysis if you do not run the script.

Access the Script

Use the script located on Stack Overflow, at https://stackoverflow.com/questions/30547283/submit-to-app-store-issues-unsupported-architecture-x86/30866648#30866648 as of August 8, 2017, to strip the non-app store related architectures from your application.