vmware airwatch

VMware AirWatch Android SDK Technical Implementation Guide

Empowering your enterprise applications with MDM capabilities using the AirWatch SDK for Android

AirWatch SDK v18.7

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

Table of Contents

Chapter 1:
Getting Started with the SDK for Android

Use the SDK for Android by selecting the SDK components you want to integrate. Meet the requirements to integrate your application and the SDK. Ensure to whitelist your application so that the Workspace ONE UEM system recognizes the application and identifies its integration with the SDK.

Component Descriptions

The SDK for Android has three components: the Client, the AWFramework, and the NetworkLibrariesSDK. Each component build upon the next. For descriptions of each components, see Introduction to the AirWatch SDK for Android on page 5.

Select the components to integrate with your application by choosing the SDK features you want to use. Review a lists of the component matched with their respective features in Compare Components on page 5.

Requirements

View a list of required components and systems to use the AirWatch SDK for Android in Requirements on page 6.

Migrate Your SDK Version

Add the necessary libraries and add dependencies to Gradle. Also ensure that the base classes have the latest code. Select the migration method based on if you use the login module. For information, see Migrate to the Latest AirWatch SDK for Android on page 6.

Whitelist Applications

Whitelist the signing key for your application with the Workspace ONE UEM system so that you can use the SDK APIs. For explanations of how to whitelist internal and public applications, see Internally Deployed Applications on page 8 and Publicly Deployed Applications on page 9.

Introduction to the AirWatch SDK for Android

The VMware AirWatch® Software Development Kit™ (SDK) for Android allows you to enhance your enterprise applications with MDM capabilities. You can use VMware Workspace ONE™ UEM features that add a layer of security and business logic to your application.

The Android SDK has several components or library sets.

SDK Library	Description
Client SDK	The client SDK is a lightweight library for retrieving basic management and device information such as compromised status, environment info, and user information.
AWFramework	The AWFramework includes an involved library for more advanced SDK functionality such as logging, restrictions, and encryption functions. The framework depends on the client SDK.
AWNetworkLibrary	The AWNetworkLibrary provides advanced SDK functionality such as application proxy and tunneling and integrated authentication. It depends on the AWFramework.

Compare Components

The SDK component you use dictates what features you can add to your applications.

For example, apps with basic MDM functionality, can use the Client SDK and omit importing the AWFramework or the AWNetworkLibrary. Whatever features used, your application must integrate the corresponding library.

SDK Component	Available Features
Client SDK	• Enrollment Status • User Info • Partial SDK Profile • Compromised / Root Detection
AWFramework	• Authentication • Client-Side Single Sign On • Branding • Full SDK Profile Retrieval • Secure Storage • Encryption • Copy Restriction

SDK Component	Available Features
AWNetworkLibrary	• Application Tunneling • NTLM and Basic Authentication • Certificate Authentication

Requirements

You must have the following systems and knowledge to use the components of the AirWatch SDK for Android.

- Android 4.4+ / KitKat
- Android API level 15+
- Android Studio with the Gradle Android Build System (Gradle) 3.3.0+
- AirWatch Agent v5.3+ for Android or Workspace ONE 3.0+
- Workspace ONE UEM console v9.2+

Emulators and Testing SDK-Built Applications

The SDK does not support testing in an emulator.

Migrate to the Latest AirWatch SDK for Android

When you migrate to the latest AirWatch SDK for Android, you must add the necessary libraries and add dependencies to Gradle. Also ensure that the base classes have the latest code.

Note: The dependent libraries packaged with the SDK are upgraded in newer versions of the SDK. When migrating, upgrade the dependent libraries with the SDK.

Migrate from 17.x

The SDK for Android does not require entries to migrate from 17.x.

Migrate from 16.10

Add the following entry to the build.gradle file.

```
android {
    defaultConfig {
    …
    ndk {
```

```
            abiFilters "x86", "armeabi-v7a", "armeabi"
        }
    }
}
```

Updates for the AWNetworkLibrary

The AirWatch SDK for Android 17.x removes the requirement to send an authentication token in an HTTP header. See Use the AWNetworkLibrary on page 34 for updated requirements.

Migrate from 16.x

Select a migration process based on the use of a login module for initialization.

Login Module

To upgrade the master key manager, override the getPassword method in the application class. This override extends AWApplication to handle the upgrade.

```
@Override
public byte[] getPassword() {
    if (SDKKeyManager.getSdkMasterKeyVersion(context)
        != SDKKeyManager.SDK_MASTER_KEY_CURRENT_VERSION) {

        SDKKeyManager.newInstance(context);
    }
    return super.getPassword();
}
```

No Login Module

Initialize your SDKContextManager and call the updateSDKForUpgrade() API.

```
SDKContextHelper sdkContextHelper = new SDKContextHelper();
SharedPreferences securePreferences = SDKContextManager.getSDKContext().getSDKSecurePreferences();
int oldSDKVersion = securePreferences.getInt(SDKSecurePreferencesKeys.CURRENT_FRAMEWORK_VERSION, 0);
SDKContextHelper.AWContextCallBack callBack = new SDKContextHelper.AWContextCallBack() {
    @Override
    public void onSuccess(int requestCode, Object result) {
        //success continue
    }

    @Override
    public void onFailed(AirWatchSDKException e) {
        //failed
```

```
    }
};
try {
    sdkContextHelper.updateSDKForUpgrade(
                    0,
                    oldSDKVersion,
                    callBack);
} catch (AirWatchSDKException e){
        //handle exception
}
```

Migrate from a Version Older Than 16.x

- Libraries
  - A total of 23 libraries including JAR and AAR files
  - SQLCipher library is an AAR file instead of a JAR file
- Gradle Methods
  - For 16.02 – compile (name:'AWFramework 16.02',ext:'aar')
  - For 16.04 – compile (name:'AWFramework 16.04',ext:'aar')
  - For both – compile (name:'sqlcipher-3.5.2-2',ext:'aar')
- Code

  If you are not using the login module for initialization, check the implementation of base classes.

For a list of the base classes that you migrate for the latest release of the AirWatch SDK for Android, see the following Workspace ONE UEM Knowledge Base article: https://support.workspaceone.com/articles/115001676868.

Internally Deployed Applications

For applications that are deployed internally, either during production or testing, the system takes the following steps to establish trust.

1. (Optional) Sign an APK file with the debug keystore of Android Studio.

   This step allows the system to whitelist the app while debugging.
2. Upload the APK file to the Workspace ONE UEM console and assign an SDK profile to the application.

   You must assign an SDK profile to the application in the Workspace ONE UEM console.
3. The Workspace ONE UEM console extracts the public signing key of the application.
4. The Workspace ONE UEM console whitelists the signing key with the AirWatch Agent or the AirWatch Container.

5. The application calls the AirWatch SDK.
6. The AirWatch Agent or the AirWatch Container validates the signing key by comparing it to the one uploaded in the Workspace ONE UEM console.

Side-Load Newer Versions for Development

After an application downloads and installs through the AirWatch Agent, then you can side-load the newer development versions signed with the same key.

Publicly Deployed Applications

For applications that are deployed publicly through the Play Store, send the public signing key of the application to Workspace ONE UEM for whitelisting.

> **Note:** Contact your professional services representative for the process of whitelisting the public signing key.

The Workspace ONE UEM system follows the same process as the internally deployed applications process to establish trust.

Chapter 2:
Integrate the Client SDK

To integrate the client SDK with your Android application, import libraries, set up your Gradle environment, set up the application to receive notifications from the Workspace ONE UEM console, and initialize in the SDKManager class.

Import Libraries

For a brief list of the libraries to import for the client SDK and information on multi-dexing, see Import the Libraries on page 10.

Set Up Gradle

Set up Gradle in Android Studio for build automation. Find out where to create the dependencies block in your Gradle file and view sample code in Set Up Gradle on page 11.

Set Up the Broadcast Receiver

Extend the AirWatchSDKBaseIntentService class to set the SDK within the application to receive notifications from the Workspace ONE UEM console. For steps and sample code, see Implement the Client SDK Broadcast Receiver on page 12.

Initialize with the SDKManager Class

Use the SDKManager class and ensure that it initializes with the application context on a background thread. For sample code, see Initialize the Client SDK on page 13.

Import the Libraries

In your project file directory, ensure that the listed files are in the libs folder.

- AirWatchSDK AAR
- GSON JAR

Multidex

When including the AirWatch SDK, it is possible your app method count may exceed 65k due to the library dependencies. In this case, enable multidex to manage the additional DEX files and the code they contain.

To enable multidex, follow the Android Developer guidelines, which you can find at the following location (as of April 2018), http://developer.android.com/tools/building/multidex.html#mdex-gradle.

Set Up Gradle

1. Ensure that your top-level build file has a classpath pointing to Gradle 3.0.0+.
2. Add a repositories block.

```
repositories{
    flatDir{
        dirs 'libs'
    }
}
```

3. To link the SDK AAR and the appropriate support library, create the dependencies block in your app-level Gradle file like the following block.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile (name: 'AirWatchSDK-18.7',ext : 'aar')
}
```

An example of the dependencies block looks like the following:

```
android {
    compileSdkVersion 26
    buildToolsVersion "26.0.1"
    defaultConfig {
        //Replace your package name here
        applicationId "<packagename>"
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        multiDexEnabled true
    }
```

```
}
// Add Google repository
repositories {
    ...
    maven {
        url "https://maven.google.com"
    }
}
```

Implement the Client SDK Broadcast Receiver

The AirWatch SDK receives commands from the Workspace ONE UEM console through the implementation of a class which extends the **AirWatchSDKBaseIntentService**.

1. Register the receiver.

   In order for your SDK app to listen for these commands, register the receiver in your Android Manifest file. You can do that by adding the following excerpt to your manifest.

```
<uses-permission android:name="com.airwatch.sdk.BROADCAST" />
<receiver
    android:name="com.airwatch.sdk.AirWatchSDKBroadcastReceiver"
    android:permission="com.airwatch.sdk.BROADCAST" >
    <intent-filter>
        <!-- Replace your app package name here -->
        <action android:name="<packagename>.airwatchsdk.BROADCAST" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.airwatch.intent.action.APPLICATION_CONFIGURATION_CHANGED" />
        <data android:scheme="app" android:host=<packagename> />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.PACKAGE_ADDED" />
        <action android:name="android.intent.action.PACKAGE_REMOVED" />
        <action android:name="android.intent.action.PACKAGE_REPLACED" />
        <action android:name="android.intent.action.PACKAGE_CHANGED" />
        <action android:name="android.intent.action.PACKAGE_RESTARTED" />
        <data android:scheme="package" />
    </intent-filter>
</receiver>
```

2. Receive the callback methods.

   Create a class named **AirWatchSDKIntentService** which extends **AirWatchSDKBaseIntentService** in the app package path to receive the callback methods.

```
package com.sample.airwatchsdk;

import ...

public class AirWatchSDKIntentService extends AirWatchSDKBaseIntentService {

    @Override
    protected void onApplicationConfigurationChange(Bundle applicationConfiguration) {
    }

    @Override
    protected void onApplicationProfileReceived(Context context, String profileId,
           ApplicationProfile awAppProfile) {
    }

    @Override
    protected void onClearAppDataCommandReceived(Context context, ClearReasonCode reasonCode) {
    }

    @Override
    protected void onAnchorAppStatusReceived(Context context, AnchorAppStatus awAppStatus) {
    }

    @Override
    protected void onAnchorAppUpgrade(Context context, boolean isUpgrade) {
    }
}
```

3. Register the intent service in your manifest.

```
<service android:name="<packagepath>.AirWatchSDKIntentService" />
```

Initialize the Client SDK

The entry point into the client SDK is the **SDKManager** class. Applications that also integrate the AW Framework do not need explicit SDKManager initialization. If you integrate the AWFramework, skip this task.

Important: It must initialize with the application context on a background thread.

The code is an example of initialization.

```
new Thread(new Runnable() {
    public void run() {
       try {
            awSDKManager = SDKManager.init(getApplicationContext());
        } catch (AirWatchSDKException e) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    String reason = "AirWatch SDK Connection Problem. " +
                                    "Please make sure AirWatch MDM Agent is Installed";
                    Toast.makeText(getApplicationContext(), reason, Toast.LENGTH_LONG).show();
                }
            });
        }
    }
}).start();
```

Once initialization completes, you can use the Client SDK.

Note: Reference the Javadoc for more in-depth information on what APIs are available.

Chapter 3:
Integrate the AWFramework

The AWFramework builds upon the Client SDK piece so setup the Client SDK.

Import Libraries

To import the libraries for the AWFramework, move the content in the Dependencies folder in the SDK ZIP file to the Libs folder of your project. For more details , see Import the Libraries and Set Up Gradle on page 16.

Set Up Gradle

Set up Gradle in Android Studio for build automation of the AWFramework. For details of the blocks to add to your project, see Import the Libraries and Set Up Gradle on page 16.

Initialize with the AWApplication Class

You can extend the AWAplication class to set application level authentication to initialize your application. For more details, see Initialize the AWFramework on page 18.

Copy and Paste Restrictions

To use the copy and paste restriction, use the available APIs. For the list, see APIs for Copy and Paste Restrictions on page 23.

Dynamic Branding

Use dynamic branding for applications that serve multiple brands with a single login page. For information, see Configure Dynamic Branding on page 24.

Push Notifications with Google Cloud Messaging

If you use Google Cloud Messaging (GCM), you can send push notifications from applications uploaded to the Workspace ONE UEM console to Android devices through GCM. For information, see Send Push Notifications Through Internal Applications with GCM and Workspace ONE UEM on page 25 and Code Push Notifications in the Application on page 26.

Import the Libraries and Set Up Gradle

Inside the SDK zip folder, move all the files located in the **Libs > AWFramework > Dependencies** folder into the libs folder for your application project.

Use the Custom, Packaged Libraries

Use the libraries provided in the SDK package. These libraries are custom and packaged to work with the SDK. The SDK does not work as designed if you replace the libraries with publicly available ones.

Set Up Gradle

Add the dependencies in your app-level Gradle build file. View the sample application for examples of an SDK file built with Gradle.

1. Add the JAR and AAR files to the dependencies section, ensuring to change the names to match the names and versions of the library files.

```
def supportLibraryVersion   = “26.1.0”
dependencies {
   compile fileTree(dir: ‘libs’, include: [‘*.jar’])
   compile "com.android.support:multidex:1.0.2"
   //Integrate with ClientSDK:
   compile (name:‘AirWatchSDK-18.7’, ext:‘aar’)
   //Integrate with AWFramework:
   compile (name:‘CredentialsExt-18.7’, ext:‘aar’)
   compile (name:‘AWFramework-18.7’, ext:‘aar’)
   compile (name:‘SCEPClient-18.7’, ext:‘aar’)
   compile "com.google.android.gms:play-services-safetynet:11.4.2"
   compile "com.android.support:support-v13:${supportLibraryVersion}"
   compile "com.android.support:appcompat-v7:${supportLibraryVersion}"
   compile "com.android.support:cardview-v7:${supportLibraryVersion}"
   compile "com.android.support:recyclerview-v7:${supportLibraryVersion}"
   compile "com.android.support:design:${supportLibraryVersion}"
   compile "com.mixpanel.android:mixpanel-android:4.+"
   compile "com.android.support:preference-v14:${supportLibraryVersion}"
```

```
    compile "net.zetetic:android-database-sqlcipher:3.5.7@aar"
}
```

2. Add a packagingOptions block with these exclusions.

```
packagingOptions {
    exclude 'META-INF/LICENSE.txt'
    exclude 'META-INF/NOTICE.txt'
}
```

3. Add a dexOptions block with these values.

```
dexOptions {
    jumboMode = true
    preDexLibraries false
    javaMaxHeapSize "4g"
}
```

4. Set the compileSDK to 26, the build tools to 26.0.1, and the targetSDKVersion to 25, all of which reside in the defaultConfig block.

```
def compileSdk = 26
def buildTools = "26.0.1"
defaultConfig {
    minSdkVersion 14
    targetSdkVersion 25
    multiDexEnabled true
    vectorDrawables.useSupportLibrary = true
    //to force fw to be merged with app when the aar included
    consumerProguardFiles file('proguard.cfg')
    ndk {
        abiFilters "x86", "armeabi-v7a"//, "armeabi"
    }
}
```

If the application needs to support ARMv6 devices (devices older than Ice-Cream Sandwich), then `armeabi` is required. For applications that only support ARMv7 devices (devices newer than Jelly Bean), it is not required. If you leave it commented out or you remove it, then the armeabi binaries are not included. Removing the armeabi binaries reduces the size of the APK by around 5 MB.

5. Sync your project with the Gradle files.

Initialize the AWFramework

The application can use application level authentication or not for initialization with the AWFramework.

Latest versions of the SDK automatically intialize both the context and gateway so you do not have to manually intialize the VMware Tunnel.

Application Level Authentication

Create a class which extends AWApplication and overrides applicable methods.

1. Create a class that extends the AWApplication class to pass configuration keys to the login module, and override the getMainActivityIntent() and getMainLauncherIntent() methods in the extended class. Move your onCreate() business logic to onPostCreate().

   The onSSLPinningValidationFailure() and onSSLPinningRequestFailure() callbacks are called when SSL pinning validation fails for the backend server (denoted by "host"). If the application does not use SSL pinning, you can leave these callbacks empty.

```
public class AirWatchSDKSampleApp extends AWApplication {

    /**
    * This method must be overridden by application.
    * This method should return Intent of your application main Activity
    *
    * @return your application's main activity(Launcher Activity Intent)
    */
    @Override
    public Intent getMainLauncherIntent() {
        return new Intent(getAppContext(), SDKSplashActivity.class);
    }

    @Override
    protected Intent getMainActivityIntent() {
        Intent intent =  new Intent(getApplicationContext(), MainActivity.class);
        return intent;
    }

    @Override
    public void onPostCreate() {
            super.onPostCreate();
            // App code here
    }

    @Override
```

```
    public void onSSLPinningValidationFailure(String host, @Nullable X509Certificate
x509Certificate) {
    }

    @Override
    public void onSSLPinningRequestFailure(String host, @Nullable X509Certificate
x509Certificate) {
    }
}
```

Optional Methods to Override

Override the methods to enable Workspace ONE UEM functionality in the AWApplication class.

- getScheduleSdkFetchTime()- Override this method to change when the login module fetches updates to SDK settings from the Workspace ONE UEM console.
- getKeyManager() – Override this method to a value rather than null so that the login module initializes another key manager and not its own.

2. In the manifest header file, declare tools.

```
<?xml version  = "1.0" encoding = "utf-8"?>
<manifest xmls:android = http://schemas.android.com/apk/res/android
    package = "<your app package name>"
    xmlns:tools = "http://schemas.android.com/tools">
```

3. Declare the tools:replace flag in the application tag that is in the manifest.

```
<application
    android:name  = ".AirWatchSDKSampleApp"
    android:allowBackup = "true"
    android:icon = "@mipmap/ic_launcher"
    android:label "@string/app_name"
    android:suportRtl = "true"
    android:theme = "@style/AppTheme"
    tools:replace = "android:label">
```

4. Set the **SDKSplashActivity** as your main launching activity in the application tag.

```
<activity
    android:name="com.airwatch.login.ui.activity.SDKSplashActivity"
```

```
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

5. In the AndroidManifest, add the certificate pinning meta tags under the opening application tag.

```
    <meta-data android:name="com.airwatch.certpinning.refresh.interval" android:value="1"/>
    <meta-data android:name="com.airwatch.certpinning.refresh.interval.unit"
android:value="DAYS"/>
```

Here is an example of the launching activity and the certificate pinning code.

```
<application
    android:name=".AirWatchSDKExampleApp"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:replace = "android:label">

    <meta-data android:name="com.airwatch.certpinning.refresh.interval" android:value="1"/>
    <meta-data android:name="com.airwatch.certpinning.refresh.interval.unit"
android:value="DAYS"/>

        <receiver
        android:name="com.airwatch.sdk.AirWatchSDKBroadcastReceiver"
        android:permission="com.airwatch.sdk.BROADCAST">
        <intent-filter>
            <action android:name="<packagename>.airwatchsdk.BROADCAST" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.PACKAGE_ADDED" />
            <action android:name="android.intent.action.PACKAGE_REMOVED" />
            <action android:name="android.intent.action.PACKAGE_REPLACED" />
            <action android:name="android.intent.action.PACKAGE_CHANGED" />
            <action android:name="android.intent.action.PACKAGE_RESTARTED" />
            <data android:scheme="package" />
```

```
        </intent-filter>
    </receiver>

    <activity
        android:name="com.airwatch.login.ui.activity.SDKSplashActivity "
        android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <activity android:name=".MainActivity" />

</application>
```

6. Add the sdkBranding to the application theme. The system displays this logo on the login screen. You can also use your own icon located in the mipmap directory.

```
<style name="SDKBaseTheme" parent="Theme.AppCompat.Light">
    // Replace with your own app specific resources to have branding
    <item name="awsdkSplashBrandingIcon">@drawable/awsdk_test_icon_unit_test</item>
    <item name="awsdkLoginBrandingIcon">@drawable/awsdk_test_icon_unit_test</item>
    <item name="awsdkApplicationColorPrimary">@color/color_awsdk_login_primary</item>
</style>
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

7. If you need the SDK authentication, DLP, and timeout behavior, app activities should extend from SDKBaseActivity. These activities allow the application to handle the lifecycle correctly and to manage the state of the SDK.

Run a Process Before Initialization, Optional

To run a process before initialization in your SDK-built application, edit the AndroidManifest.xml file and customize an activity.

Use this optional procedure to run processes that analyze the environment into which the application is deployed. For example, run a process to determine if your application needs to start the SDK in a specific environment.

1. In the AndroidManifest.xml file, remove the launcher tag `<category android:name="android.intent.category.LAUNCHER" />`.

   You add the tag in the placeholder activity you create to run your process.

2. Create an activity and register it in the AndroidManifest.xml file.

   This activity runs the desired process.

3. Add the intent filter to the activity you just created in the manifest. The filter resembles the example.

```
<intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

4. Call `startActivity(new Intent(this, SDKSplashActivity.class))` after the process completes.

Use the AWFramework

Add code and use APIs to configure AWFramework capabilities in your application.

Retrieve the SDK Profile

Once the SDKContext is in the configured state, you can call the SDKContext getSDKConfiguration() method to retrieve the SDK profile. The SDK must be finished with its configuration otherwise calling getSDKConfiguration() returns with an empty value.

```
if (sdkContext.getCurrentState() == SDKContext.State.CONFIGURED) {
    String sdkProfileString = SDKContextManager.getSDKContext().
    getSDKConfiguration().toString();
}
```

Encrypt Custom Data

Once the SDKContext is in the initialized state, you can call the data encryption API set. This set of functions uses the AirWatch SDK's intrinsic key management framework to encrypt and decrypt any data you feed in.

Use the MasterKeyManager API set when the SDK is in an initialized or configured state. See the example of how you can encrypt and decrypt a string value.

```
if (SDKContextManager.getSDKContext().getCurrentState()!=SDKContext.State.IDLE) {
    MasterKeyManager masterKeyManager = SDKContextManager.getSDKContext().getKeyManager();
    String encryptedString = masterKeyManager.encryptAndEncodeString("HelloWorld");
```

```
    String decryptedString = masterKeyManager.decodeAndDecryptString(encryptedString);
}
```

Secure Storage Data

After the SDKContext is in the initialized state, you can call the secure storage API set. This set of functions stores key value pairs in encrypted storage.

```
if (SDKContextManager.getSDKContext().getCurrentState()!=SDKContext.State.IDLE) {
    SecurePreferences pref = SDKContextManager.getSDKContext().getSDKSecurePreferences();
    pref.edit().putString(<KEY_NAME>, <VALUE>).commit(); // to store value
    Object value = pref.getString(<KEY_NAME>, <Default_Value>);
}
```

APIs for Copy and Paste Restrictions

To use the AirWatch SDK copy restriction, replace the Android classes in your application to the listed Workspace ONE APIs.

Examples

If Java class `XYZ` extends `TextView{...}`, change it to extend `AWTextView{...}`.

If you override the method `onTextContextMenuItem(int id)`, do not process the listed IDs. You must call `return super.onTextContextMenuItem(id);` for the listed IDs.

- `android.R.id.cut`
- `android.R.id.copy`
- `android.R.id.paste`
- `android.R.id.shareText`

Change `<TextView>` in all Layout XML or View XML to `<com.airwatch.ui.widget.AWTextView.../>`.

APIs

Android Class	AirWatch SDK API
android.support.v7.widget. AppCompatEditText	com.airwatch.ui.widget. AWEditText
android.support.v7.widget. AppCompatTextView	com.airwatch.ui.widget. AWTextView
android.support.v7.widget. AppCompatAutoCompleteTextView	com.airwatch.ui.widget. AWAutoCompleteTextView

Android Class	AirWatch SDK API
android.support.design.widget. TextInputEditText	com.airwatch.ui.widget. AWTextInputEditText
android.support.v7.widget. SearchView.SearchAutoComplete	com.airwatch.ui.widget. AWSearchAutoComplete
android.webkit.WebView	com.airwatch.ui.widget.CopyEnabledWebView

Configure Dynamic Branding

Use dynamic branding for applications that serve multiple brands with a single login page. This feature requires settings in the Workspace ONE UEM console and code changes in the application.

Console Settings

Request your Workspace ONE UEM admin to set the listed values in the console. Find settings in **Groups & Settings > All Settings > Apps > Settings and Policies > Settings > Branding**.

1. Enable **Branding**.
2. Add a color in the **Primary Color** field.

   The application uses this color for the background of splash and loading pages and for action buttons.

Application Code

Change the code to add a company logo and to implement your branding scheme.

- **Company logo** - To add a company logo for dynamic branding, override **isInputLogoBrandable()** in the **AWApplication** class.

  Example

```
/**
* Returns if the company logos are brandable.
* @return true if branded logos needs to be used on the SDK UI screens, and false if not
*/
protected boolean isInputLogoBrandable() {
    return true;
}
```

- **Branding scheme** - Add your branding scheme to **BrandingManager** and inject it into the **BrandingProvider** class.

  Example

```
public abstract class AWApplication extends MultiDexApplication implements BrandingProvider {

    private BrandingManager brandingManager;

    @Override
    public final void onCreate() {
        super.onCreate();
        brandingManager = new DefaultBrandingManager(
            SDKContextManager.getSDKContext().getSDKConfiguration(),
            new SDKDataModelImpl(getApplicationContext()),
            getApplicationContext());
    }

    @Override
    public BrandingManager getBrandingManager() {
        return brandingManager;
    }
}
```

Send Push Notifications Through Internal Applications with GCM and Workspace ONE UEM

If you use Google Cloud Messaging (GCM), you can send push notifications from applications uploaded to the Workspace ONE UEM console to Android devices through GCM. Devices must have an SDK-built application on it that supports push notifications.

Important: Firebase Cloud Messaging (FCM) replaces GCM. This outlined process does not work for applications that use FCM.

Requirement

This feature requires Workspace ONE UEM console v9.5+.

Configure GCM

To send notifications using GCM, configure GCM and get a sender ID and an API server key. These two values enable GCM, Workspace ONE UEM, the client, and the application to communicate

- **Sender ID** - The GCM system generates this value when you configure an GCM API project. The system uses the value to identify the package ID for the application.
- **API Server Key** - This key gets saved on application server to authorize the server to access Google services. Enter this value when you upload the internal application to the Workspace ONE UEM console.

The Client Gets a Registration Token from GCM

Call the registration API on the client to register with GCM. The client then sends the registration token to the Workspace ONE UEM console.

Each client has its own registration token. The console uses the registration token to identify the client for which push notifications are sent through GCM servers.

Registration Process

1. Register the application with GCM using the directions at https://developers.google.com/cloud-messaging/android/client.
2. Retrieve the sender ID and code it in the application class.
3. Retrieve the API server key and enter it in the console while uploading the app.
4. Call Register API on the client side to register the device with the GCM server and to retrieve the registration token (GCM token).
5. The SDK beacon sends the token to the Workspace ONE UEM console. If you want to force sending the token immediately, stop and restart the application.

Send Push Notifications With Workspace ONE UEM

Send push notifications from the Workspace ONE UEM console with these steps.

1. Navigate to **Apps & Books > Applications > Native > Internal** and select the application from the list view.
2. Select the **Devices** tab.
3. Select all the devices to which you want to send the notification.
4. Select **Send Message to All** or **Send**.
5. Select **Push Notification** for the **Message Type**.
6. Select the SDK-built application in the **Application Name** field the system uses to send push notifications to selected devices.
7. Enter the message in the **Message Body**, complete the rest of the procedure, and send it to devices.

Code Push Notifications in the Application

To use push notifications in your application and Workspace ONE UEM, code the support of push notifications in the AWFramework module in the application class.

Use the PushNotificationContext Interface

The AWFramework module provides an abstract AWApplication class that runs the **PushNotificationContext** with default behaviors. Your application can override some of the methods to intercept callbacks for additional actions or if it does not use the AWApplication class.

Requirements

- Code the sender ID from GCM into the application class for push notifications to work.
- Add the API server key from GCM to the Workspace ONE UEM console when the admin uploads the application.

Add Code

Follow this procedure to code support for push notifications if your application does not use the AWApplication class.

1. Override the **registerForPushNotifications()** method in the **PushNotificationContext** class and call `GCMManager.registerForPushNotifications(getApplicationContext());`.

   A call to **registerForPushNotifications()** initiates the GCM registration process.

2. Override **getSenderID()** and return your application's corresponding sender ID.
3. To intercept GCM registration and GCM push notification events, use the callbacks in **IPushNotificationReceiver**.
4. If your application sends other push notifications besides those for Workspace ONE UEM, override **AWPushNotificationReceiver** implementation and call super.

   For details about specific Workspace ONE UEM actions, review **AWPushNotificationReceiver**. Actions include uploading the registration token and checking for secure messages in the console.

   If the application class overrides AWApplication and does not use push notifications for anything else, do not override **getPushNotificationReceiver()**.

5. To code that the application received a notification and to not use AWApplication's default behavior, override **onSecureMessageReceived(String message)** with the desired behavior.

   The default behavior is to display a notification in the notification bar with a Workspace ONE UEM icon.

SCEP Support to Retrieve Certificates for Integrated Authentication

The SDK supports the SCEP protocol, with limitations, to retrieve certificates for integrated authentication. To use SCEP certificates for your SDK-built application, ensure integrated authentication is enabled and that SCEP is configured in the console as a certificate authority.

Supported SAN Information Types

The SDK fully supports the listed Subject Alternative Names (SAN) information types in certificate attributes.

- rfc822Name
- dNSName
- uniformResourceIdentifier

Supported with Correct Format

The SDK supports the listed SAN information types but you must use the correct format or the SDK ignores them.

- iPAddress
- registeredID

Not Supported

The SDK does not support the listed SAN information types. If you configure them, the SCEP process fails.

- otherName
- x400Address
- directoryName
- ediPartyName
- GUID
- Custom

APIs for a Pending Status from the SCEP Certificate Authority

Use the SCEPCertificateFetcher method to modify SCEP certificate fetches to poll the SCEP server and to refetch certificates when the server returns a pending status.

Pending Status of Certificate Fetches

When using SCEP, some configurations set the SCEP certificate authority not to issue the certificate until the request is approved. In this scenario, the authority returns a pending status to the SDK.

The SDK for Android fetches SCEP certificates when you enable integrated authentication and have SCEP configured as a certificate authority. However, if you want your application to handle the listed scenarios, you must modify code.

- You want the application to handle a pending status result for a certification fetch.
- You want the application to know the result of the fetch and to act based on the result.

Ensure the Certificate Authority Server Handles Retry Requests

The SDK retries the fetch request based on the parameters in the modified code or using the default behavior (retries every 5 milliseconds for 10 tries). If the server is not configured to handle retry requests caused by the pending status, the fetch never finishes.

Methods for Pending Status

To handle the pending result and to poll the server to refetch, modify the SCEPCertificateFetcher method.

1. Set the retry interval and maximum retry count.

```
/**
 * Sets the maximum number of retry attempts. Once this is elapsed, the pending status is
```

```
 * cleared, polling stops and the next fetch results in a new SCEP request for a new
 * certificate.
 */
void setMaxRetryCount(int maxRetryCount);

/**
 * Sets the retry interval between fetch attempts (in seconds).
 */
void setRetryIntervalSeconds(int retryIntervalSeconds);
```

2. Pause and resume the polling mechanism. Use the Activity Lifecycle methods so that the polling mechanism does not run when the application is not in the foreground.

```
/**
 * Triggers/resumes scheduled polling for fetching "Pending" certificates. To be called
 * when the application is brought to foreground(in any Activity's onResume()).
 */
void triggerPolling();

/**
 * Pause polling for SCEP pending certificates. To be called when the application is
 * sent to background (in any Activity's onPause()).
 */
boolean pausePolling();

/**
 * Returns true if a SCEP certificate fetch is pending. This will be reset to false when the
 * SCEP certificate polling results in a success or failure or if maximum number of attempts
 * is exceeded.
 */
boolean isSCEPCertificatePending();
```

3. Register listeners to identify the SCEP certificate fetch result.

```
/**
 * Register an implementation of {@link SCEPCertificateFetchListener} to listen for fetch
 * results. Ensures that one instance is added only once. Unregister the listeners using
 * {@link #unregisterFetchListener(SCEPCertificateFetchListener)} when callbacks are no longer
 * required in order to prevent memory leaks of the listener implementation.
 */
```

```
void registerFetchListener(SCEPCertificateFetchListener statusListener);

/**
 * Unregister the registered listener when callbacks are no longer necessary. This ensures
 * that the listeners are not leaked.
 */
void unregisterFetchListener(SCEPCertificateFetchListener statusListener);
```

Obtain and instance of the SCEPCertificateFetcher using `SCEPContext.getInstance().getSCEPCertificateFetcher()`.

Example of an Activity to Poll and Listen for SCEP Fetch Results

```
public class IntegratedAuthActivity extends AppBaseActivity implements View.OnClickListener{

    private static final String TAG = "IntegratedAuthActivity";

    private static final String SCEP_MAX_COUNT_KEY = "max_count";
    private static final String SCEP_RETRY_INTERVAL_KEY = "retry_interval";

    private SCEPCertificateFetchListener certStatusListener = new SCEPCertificateFetchListener() {
        @Override
        public void onResult(CertificateFetchResult certificateFetchResult) {
            handleResult(certificateFetchResult);
        }
    };

    private SCEPCertificateFetcher certificateFetcher = SCEPContext.getInstance
().getSCEPCertificateFetcher();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_ntlm_and_basic_auth);

        certificateFetcher.registerFetchListener(certStatusListener);

        final SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);

        //These values are fetched from SharedPreferences.
        int maxCount = pref.getInt(SCEP_MAX_COUNT_KEY, SCEPCertificateFetcher.DEFAULT_MAX_RETRY_
```

```
COUNT);
        int retryInterval = pref.getInt(SCEP_RETRY_INTERVAL_KEY, SCEPCertificateFetcher.DEFAULT_
RETRY_INTERVAL_SECONDS);

        certificateFetcher.setMaxRetryCount(maxCount);
        certificateFetcher.setRetryIntervalSeconds(retryInterval);
    }

    @Override
    protected void onResume() {
        super.onResume();

        if(certificateFetcher.isSCEPCertificatePending()){
            certificateFetcher.triggerPolling();
        }
    }

    @Override
    protected void onPause() {
        super.onPause();

        pauseSCEPCertificatePolling();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();

        certificateFetcher.unregisterFetchListener(certStatusListener);
    }

    private void handleResult(CertificateFetchResult result) {
        switch (result.getStatus()){
            case SUCCESS:
                showSnackbar("SCEP certificate fetch succeeded");
                break;
            case FAILURE:
                String errorString = getErrorString(result.getErrorCode());
                showSnackbar("SCEP certificate fetch failed. " + errorString);
                break;
```

```
            case PENDING:
                String messageString = getErrorString(result.getErrorCode());
                PendingRetryDataModel retryDataModel = result.getPendingRetryDataModel();
                String retryMessage = "Attempts remaining: " +
retryDataModel.getRetryAttemptsRemaining() +
                        ". Time remaining for next retry: " +
retryDataModel.getTimeRemainingForNextRetryAttempt();
                showSnackbar(messageString + " " + retryMessage);
                break;
        }
    }
}
```

Chapter 4:
Integrate the AWNetworkLibrary

The AWNetworkLibrary builds on top of the AWFramework. Integrate the Client SDK and the AWFramework to prepare for the AWNetworkLibrary.

Set Up Gradle and Initialize

The AWNetworkLibrary uses the steps that initialize the AWFramework with a login module. It takes these steps further so for more details, see Integrate the AWNetworkLibrary on page 33.

Set Up Gradle and Initialize the AWNetworkLibrary

To add the AWNetworkLibrary to your project, follow the listed process.

1. Set up Gradle.

   Add all the dependency JARS and AARS from **libs > AWNetworkLibrary > Dependencies**. For each AAR file, add an entry stating the name and EXT type.

```
dependencies {
    ... // In addition to AWFramework entries add the listed library
    compile (name:'AWNetworkLibrary-18.7', ext:'aar')
}
```

2. Initialize the AWNetworkLibrary.
   a. Follow the steps outlined in Initialize the AWFramework on page 18.
   b. In the extended AWApplication class, override getMagCertificateEnable() and return true to fetch the certificate for the VMware Tunnel.

```
/**
* This method is overriden if your application supports fetch mag certificate during login
process.
*
* @return true if app supports fetch mag certificate.
*/
@Override
public boolean getMagCertificateEnable() {
    return true;
}
```

c. Set GatewaySplashActivity as your main launching activity instead of SDKSplashActivity.

```
<activity
    android:name="com.airwatch.gateway.ui.GatewaySplashActivity"
    android:label="@string/app_name"  >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

d. Extend GatewayBaseActivity at the activity level to support network features like tunneling and integrated authentication in addition to AWFramework features.

Use the AWNetworkLibrary

The AWNetworkLibrary provides SDK capabilities like application tunneling and integrated authentication using NTLM and SSL/TLS client certificates for various HTTP clients.

The AWNetworkLibrary does not require the use of AirWatch-provided HTTP clients or WebView to tunnel requests through the VMware Tunnel Proxy. Use common HTTP clients or the default WebView. The AWNetworkLibrary also provides new APIs for NTLM or SSL/TLS integrated authentication that you can use with available HTTP client APIs. Review the tables to see what methods have changed to configure these features.

Applications using the existing provided HTTP Client and WebView classes do not require any changes.

Application Tunneling

Use any HTTP clients or the default WebView for tunneling application traffic through the VMware Tunnel Proxy.

Capability	Previous Requirements	Updated Requirement
Application Tunneling		
Tunnel HTTP request with the VMware Tunnel Proxy.	Use AWHttpClient, AWUrlConnection, or AWOkHttpClient.	Use any HTTP client.
Tunnel WebView requests with the VMware Tunnel Proxy.	Use AWWebView.	Use the default WebView.

Integrated Authentication

Use APIs with various HTTP clients for integrated authentication using the NTLM method or SSL/TLS client certificates. These APIs eliminate the need to provide HTTP clients in some cases.

Note: Developers that use the existing APIs to achieve integrated authentication functionality do not require any changes.

Note: Find code samples that use the integrated authentication APIs in the IntegratedAuthActivity.java file in the sample application.

Capability	Previous Requirements	Updated Requirement
Integrated Authentication - NTLM		
Add support for NTLM integrated authentication for HTTP clients.	Use AWHttpClient or NTLMHttpUrlConnection as wrapper classes.	• For Apache HttpClient register an NTLM AuthScheme and add an AWAuthInterceptor request interceptor. • For UrlConnection, there is no change. Use NTLMHttpUrlConnection. • For OkHttpClient, set an instance of AWOkHttpAuthenticator as an authenticator.
Add support for NTLM integrated authentication for WebViews.	Use AWWebView or AWWWebViewclient.	No change. Use one of the listed methods. • Set an instance of AWWebViewClient as the WebViewClient for the WebView. • Extend the AWWebViewClient class and customize it for several methods.

Capability	Previous Requirements	Updated Requirement
Integrated Authentication - SSL Client Certificate		
Add support for SSL client certificate authentication for HTTP clients.	Use AWHttpClient and AWUrlConnection.	Use the API called AWCertAuthUtil. It provides methods to construct an SSLContext instance with the required certificates for authentication. You can then plug it into various HTTP clients like Apache HTTP Client, URLConnection, and OKHttpClient. For example, for HttpClient, retrieve a list of KeyManagers from the API AWCertAuthUtil.getCertAuthKeyManagers(). Use this list to construct an instance of SSLContext. Obtain an instance of SSLSocketFactory from the SSLContext instance and use it in the HttpClient.
Add support for SSL client certificate authentication with WebViews.	Use AWWebViewClient.	No change. Use AWWWebViewClient. Extend the class and override unneeded behaviors.

Chapter 5:
Sideload a Debug Version of the Application for Developing

To help with developing, testing, and debugging your SDK-built applications, you can sideload them or push them from Android Studio.

However, the Workspace ONE UEM admin must have installed the same application with the same signature previously, with the AirWatch Agent or Workspace ONE.

1. The Workspace ONE UEM admin installs the SDK-built application as a signed APK with a debug key and pushes it to devices with either the AirWatch Agent or Workspace ONE.
   - The admin must use the same signing key that you used for development.
   - The admin must publish the application as managed.
2. The developer side-loads and runs the debug versions of the application, that is signed with the same key, on the device.

   The signature of the console application and the side-loaded application must match.

The Agent or Workspace ONE trusts and runs the debug version on the device.

If the application is uninstalled from the console or is uninstalled at the device, repeat this process to run a trusted debug version of the application.