Introduction to Samsung Knox ISV SDK

SRIN B2B Innovation Lab

b2b-tech-support-sea@samsung.com

Table of Contents

Preface

Introduction

Samsung KNOX SDK is a collection of API (Application Programming Interface) that enables enterprises to design applications which can manage their employee's mobile devices. Applications developed with KNOX SDK API can reduce security threats and risks from lost or stolen devices that contain sensitive corporate data. [1]

Samsung KNOX SDK includes various SDKs, including Standard SDK, Premium SDK, Customization SDK, SSO SDK, VPN SDK, ISV SDK, Samsung EDU SDK for Android operating system, Standard SDK for Tizen operating system, Cloud SDK, and various KNOX based services such as KNOX Enabled App and KNOX Customization Service.

Before you can use the Samsung Knox SDK in your app, you need to obtain a Samsung Knox license (SKL). Samsung Knox uses a license manager server to identify and authenticate apps that can take control of devices. You use your SKL key in your app for authentication purposes. You SKL key is passed to every device that you manage with your app. Should your SKL key become compromised, Samsung can revoke it, and any app that uses the compromised key can't take control of a device.

NOTE – The SKL key replaces the ELM and ISV keys that were used in the legacy Knox SDKs. If you are still using these legacy SDKs, for example, on older devices, see About license keys. Currently, for paid premium features, you still need to use Production ELM and KLM licenses. An SKL license will **not work.** This should change eventually after the SKL license is out of beta. You can generate Production ELM and KLM licenses from the SEAP Portal.

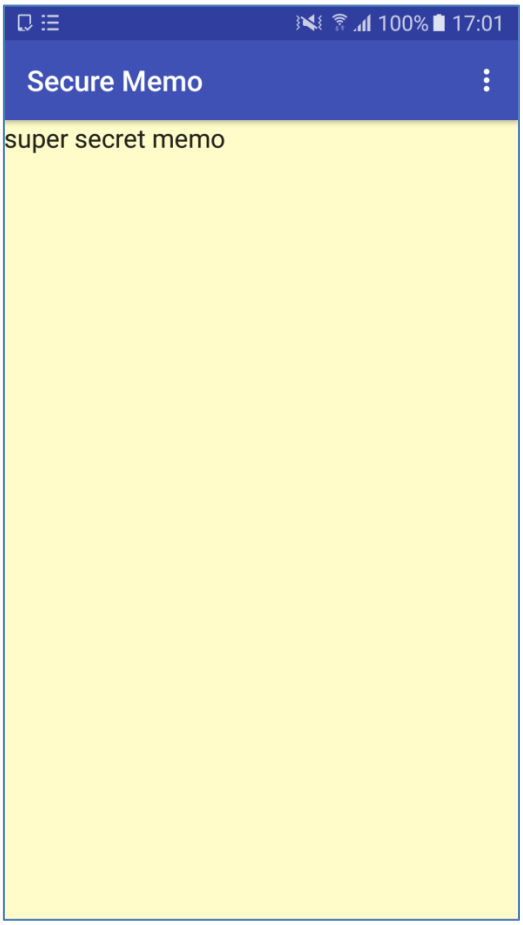
This **Introduction to Samsung KNOX ISV SDK** guide is dedicated to developers who are interested to develop enterprise applications using KNOX ISV SDK.
This guide book covers usage of Attestation and Sensitive Data Protection (SDP) SDK which are part of ISV SDK;

What Will We Create

In this book you will create Secure Memo application. There are two main parts of this sample application that will be covered:

1. Device Integrity Checking
   - Check if a device has been rooted or is running unauthorized firmware.
2. Secure Data Protection
   - Save data into secure location.

Both functions will use Knox SDK for implementation.

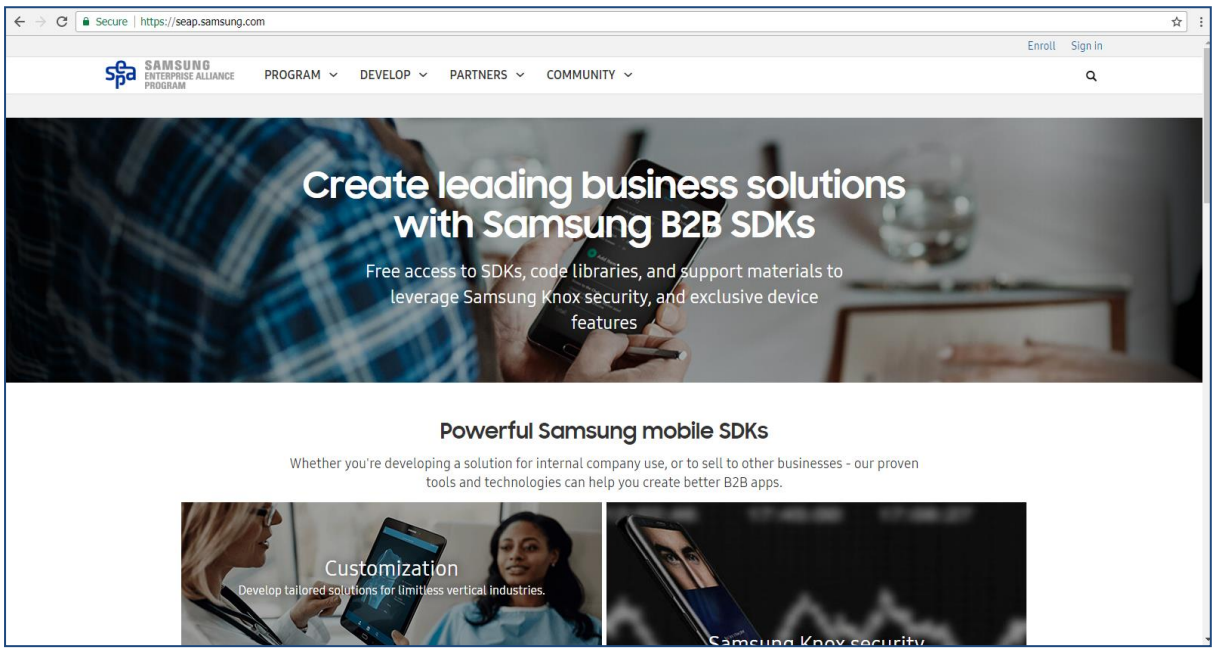


In the next part you will start to develop the application from getting license key.

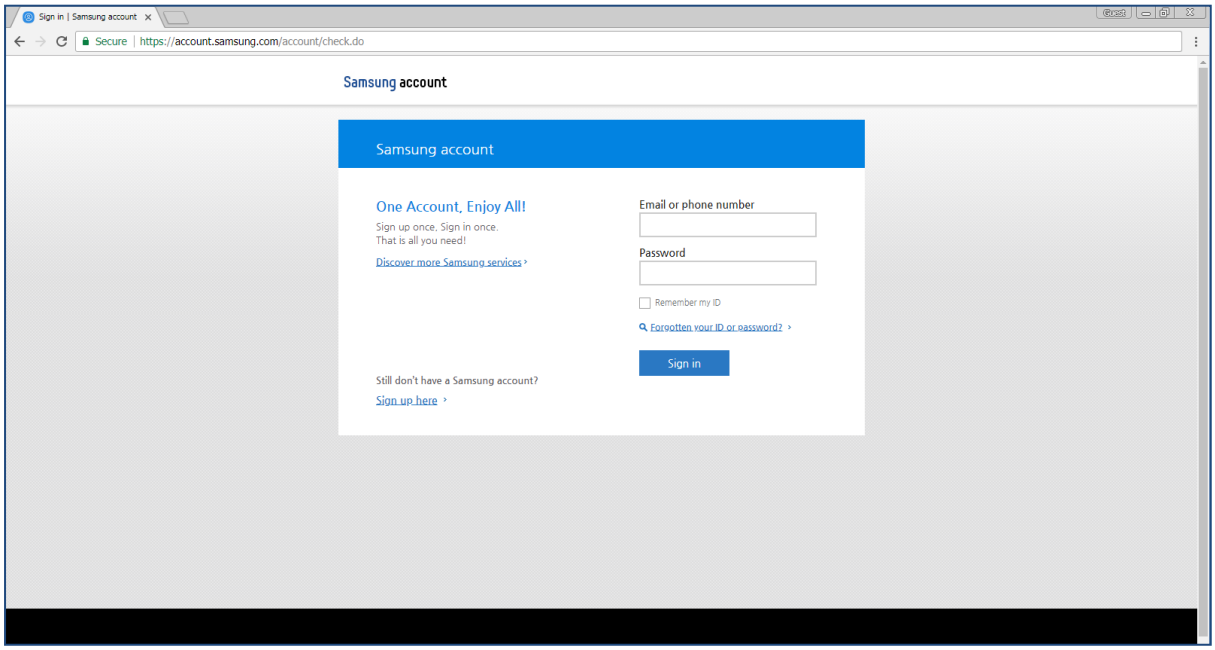
Get KNOX SDK

To obtain KNOX SDK and licenses to be used for developing applications using KNOX SDK, users need to be registered as B2B partner in the SEAP (Samsung Enterprise Alliance Program) portal.

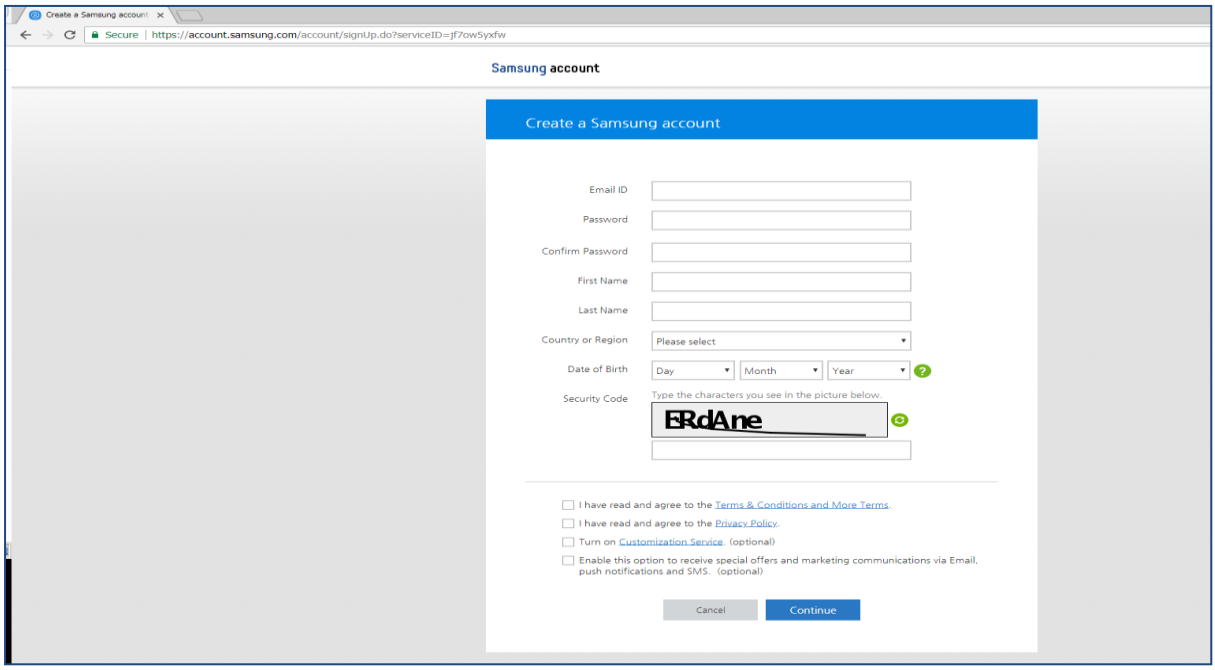
SEAP portal is accessible using this link https://seap.samsung.com/.



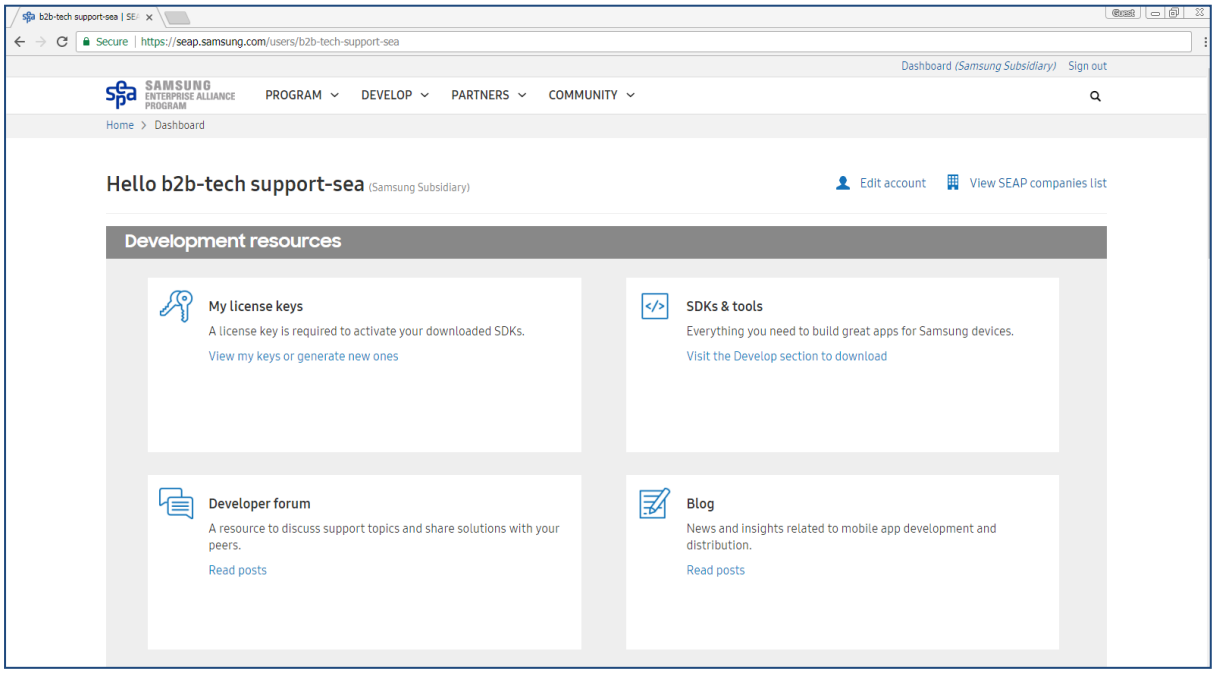
To sign in as registered SEAP partner, users can use **Sign in** menu on the SEAP portal front page. Users can use Samsung Account to log in to the SEAP portal.



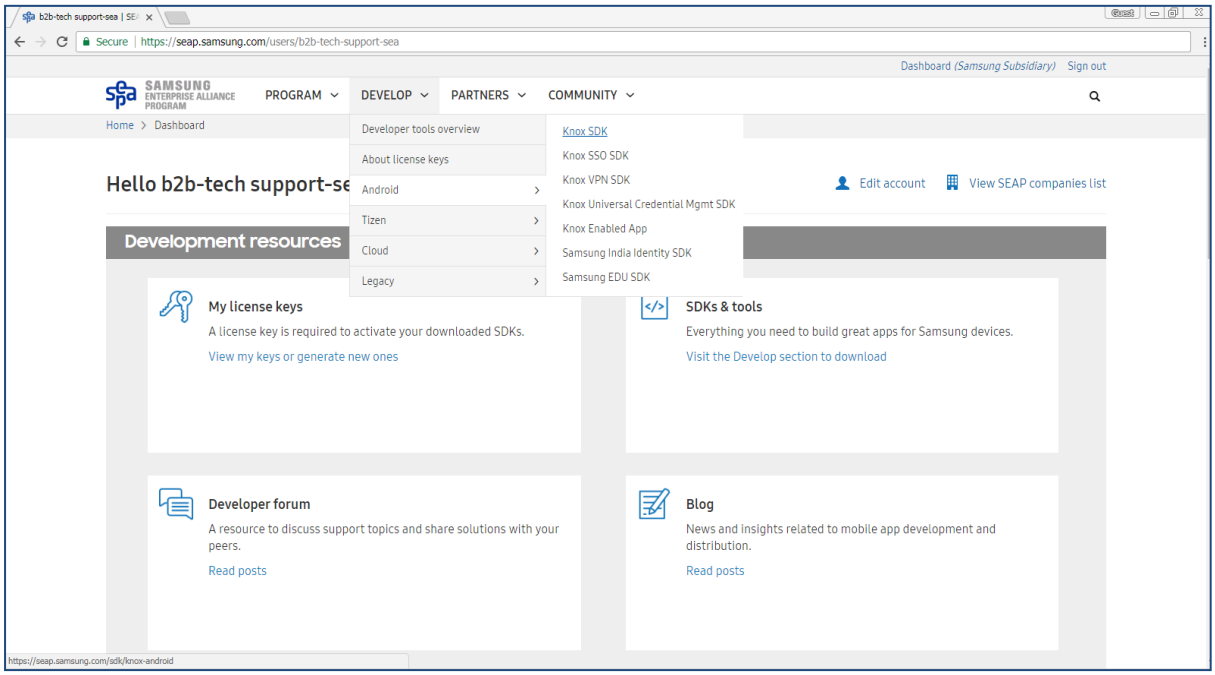
First time users using Samsung Account to log in to the SEAP portal need to enroll their accounts as B2B partner through the **SEAP Enrollment** page which is also accessible using the **Enroll** menu on the SEAP portal front page.



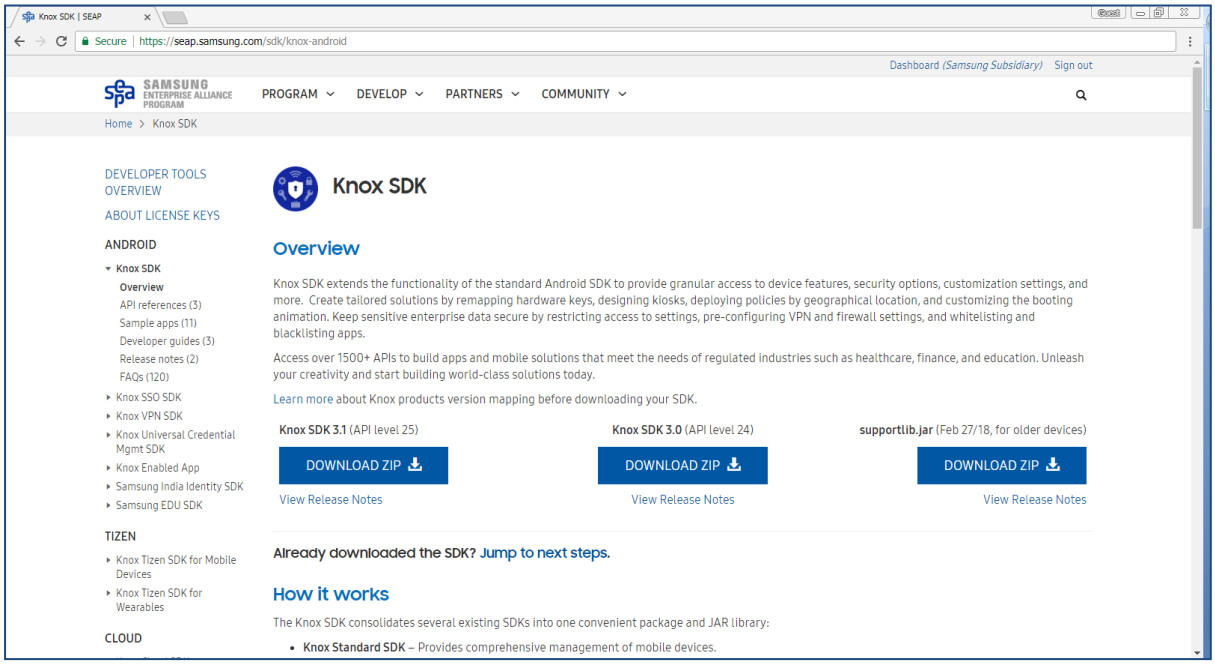
After logged in to the SEAP portal, users will be redirected to the SEAP portal main page where users can download KNOX SDK, request and view generated licenses, open developer forum, among other things.



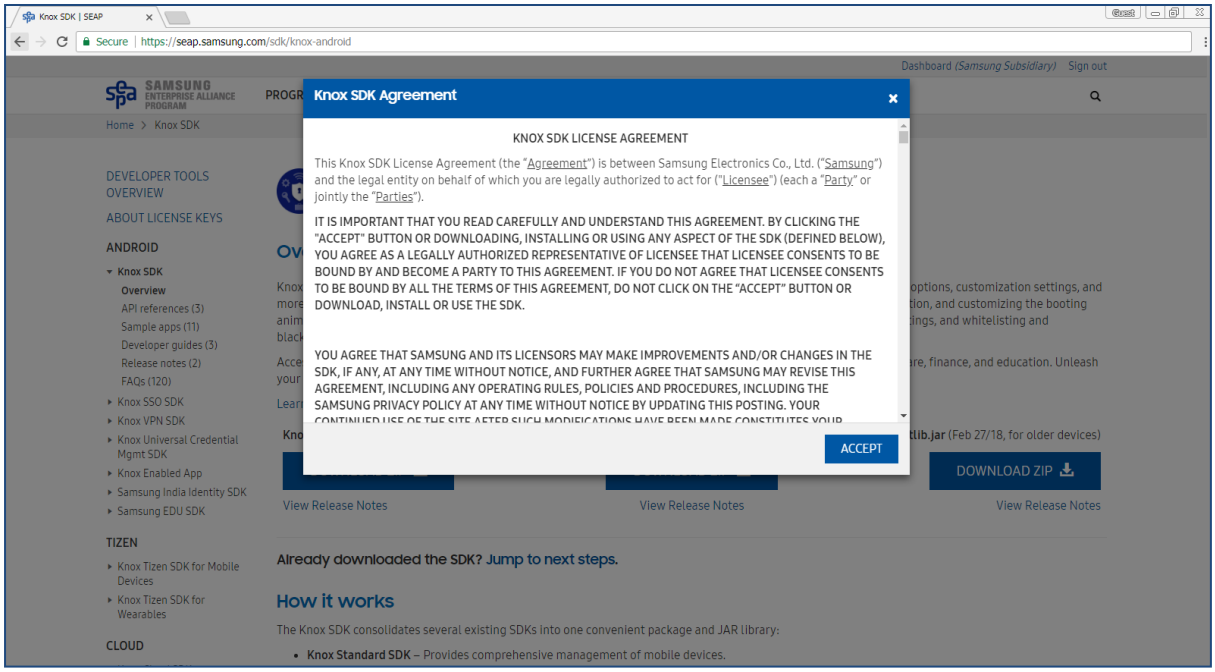
KNOX SDK can be accessed using menu that is available on the SEAP portal main page by selecting **DEVELOP → Android → KNOX SDK**.



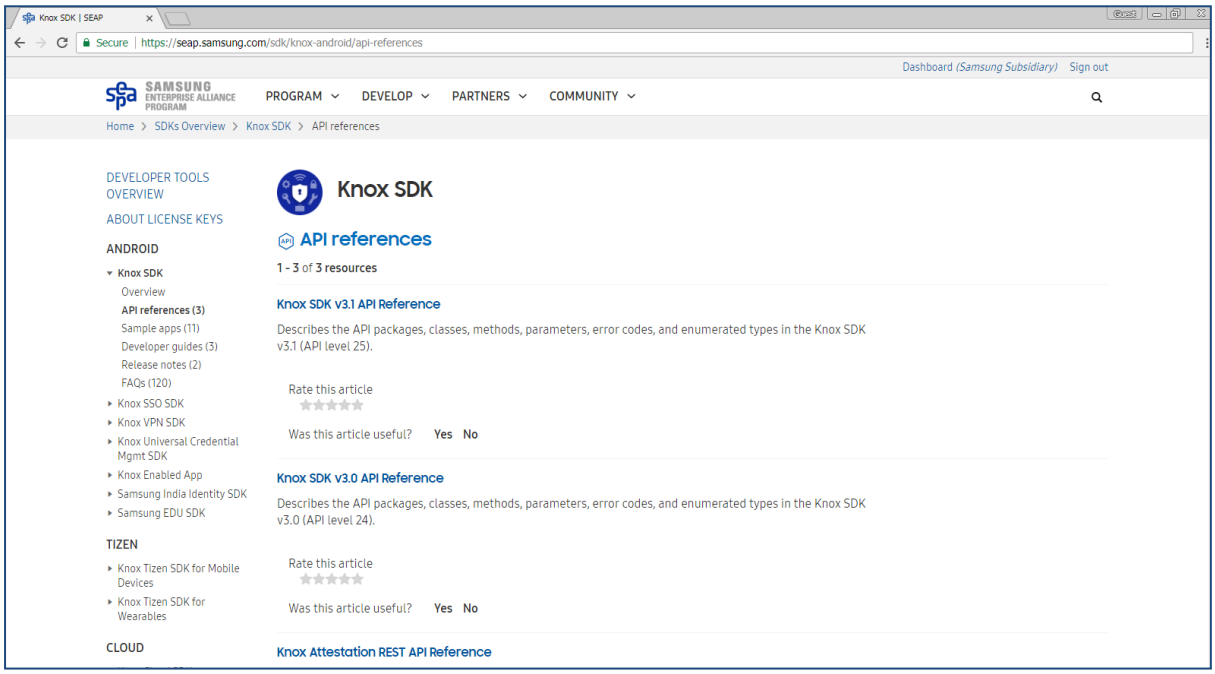
U can download KNOX SDK libraries by clicking the **DOWNLOAD SDK** button.
This step can be repeated for every KNOX SDKs to download all the KNOX SDKs available on the SEAP portal.

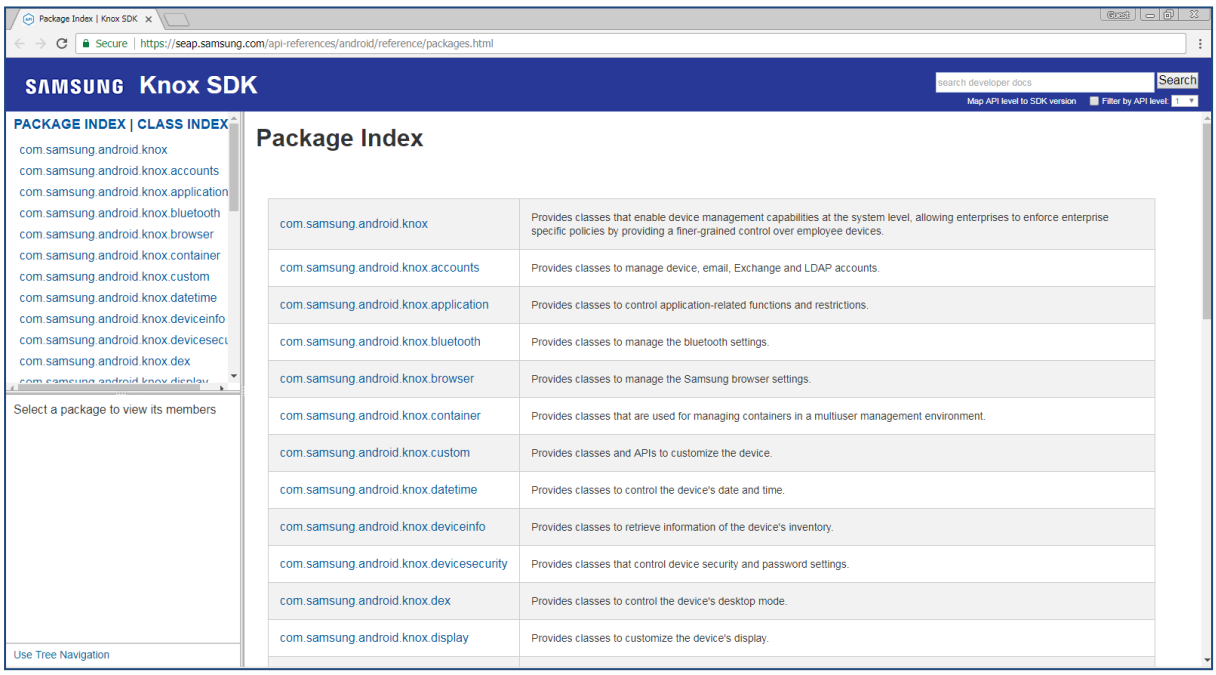


Users need to agree to the SEAP agreement to be able to download and use KNOX SDK.

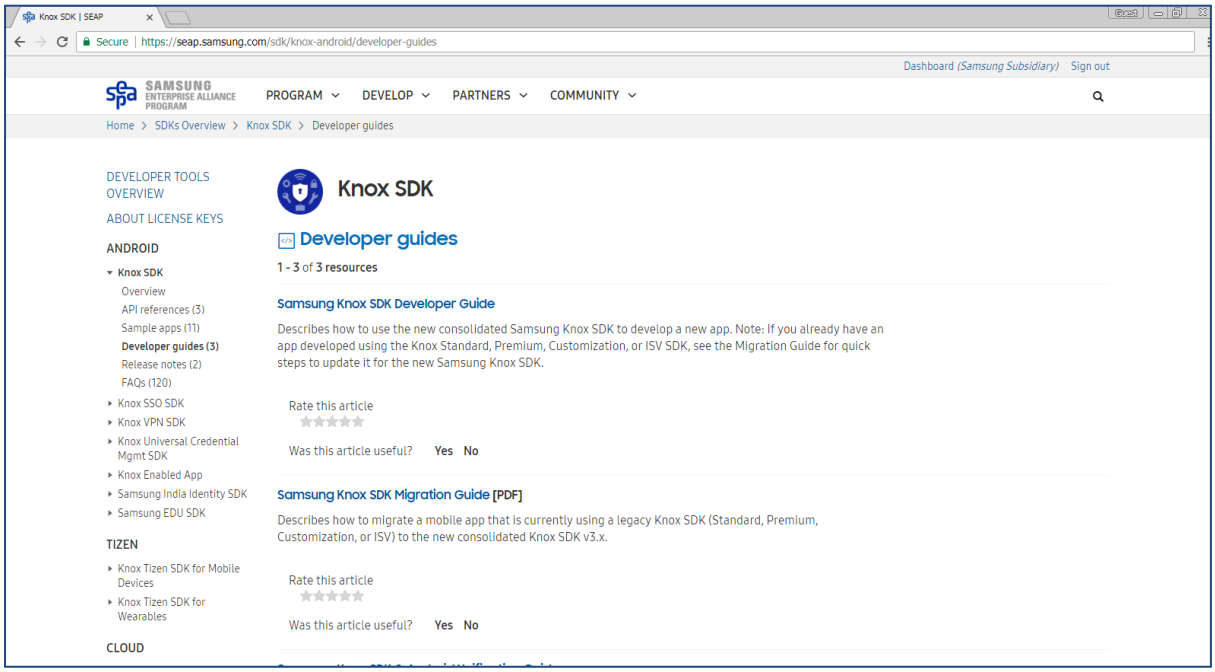


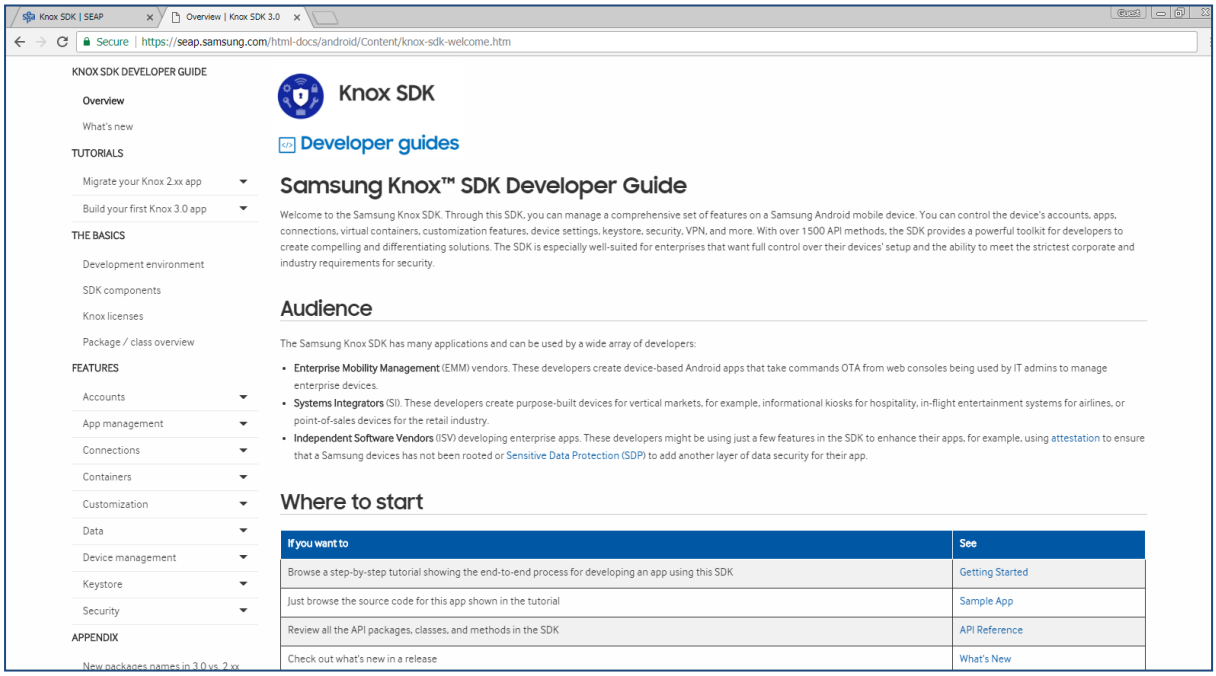
Users can also find several supporting pages to help during development on the side navigation menu.
One of which is the **API References** page where users can list all the API packages, classes, methods, parameters, error codes and enumeration types used by KNOX SDK.



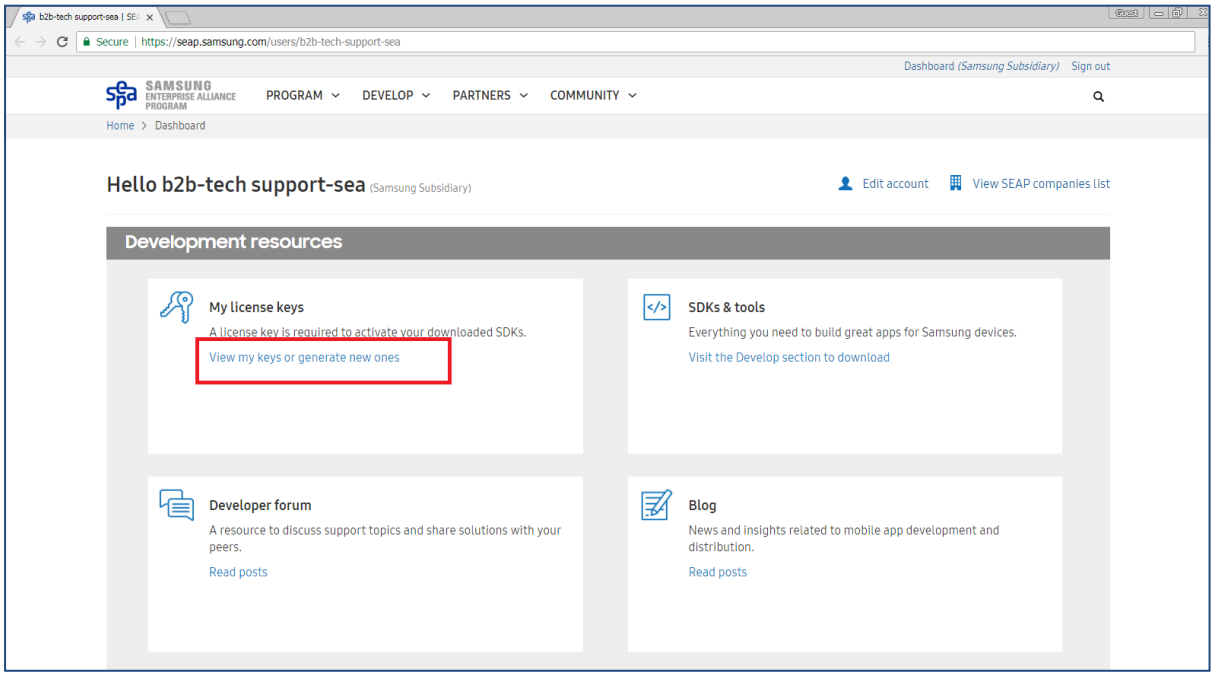


There is also **Developer Guides** page which illustrates on how to getting started with KNOX SDK development, and provides sample codes for KNOX SDK features.

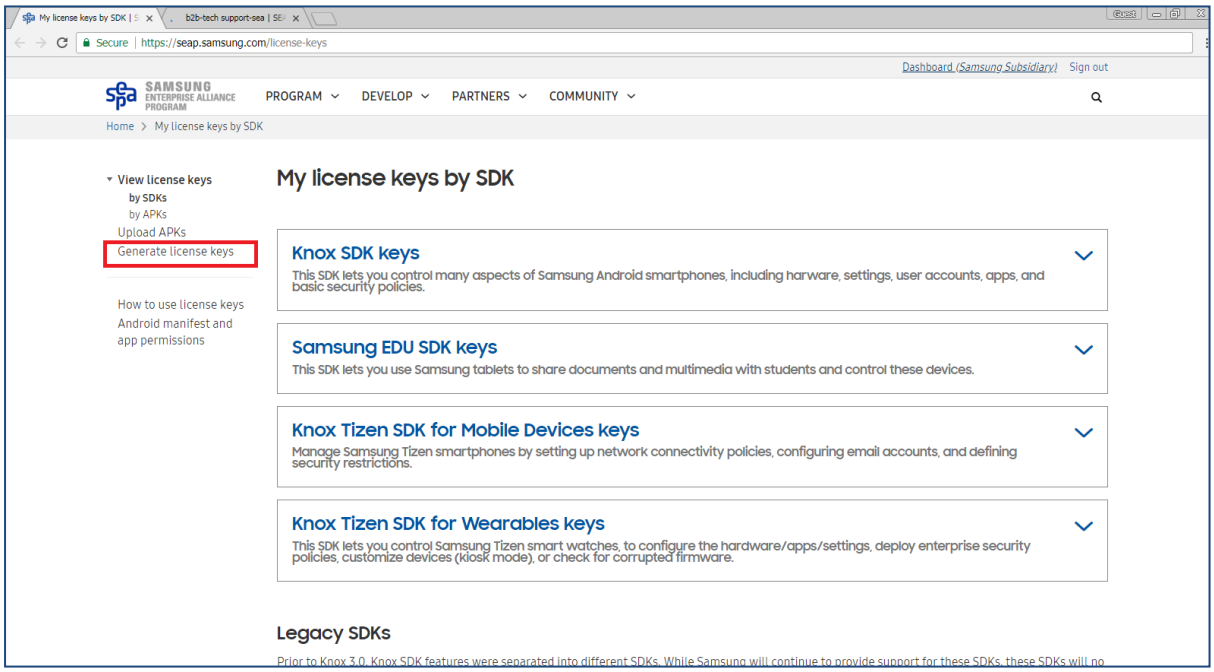


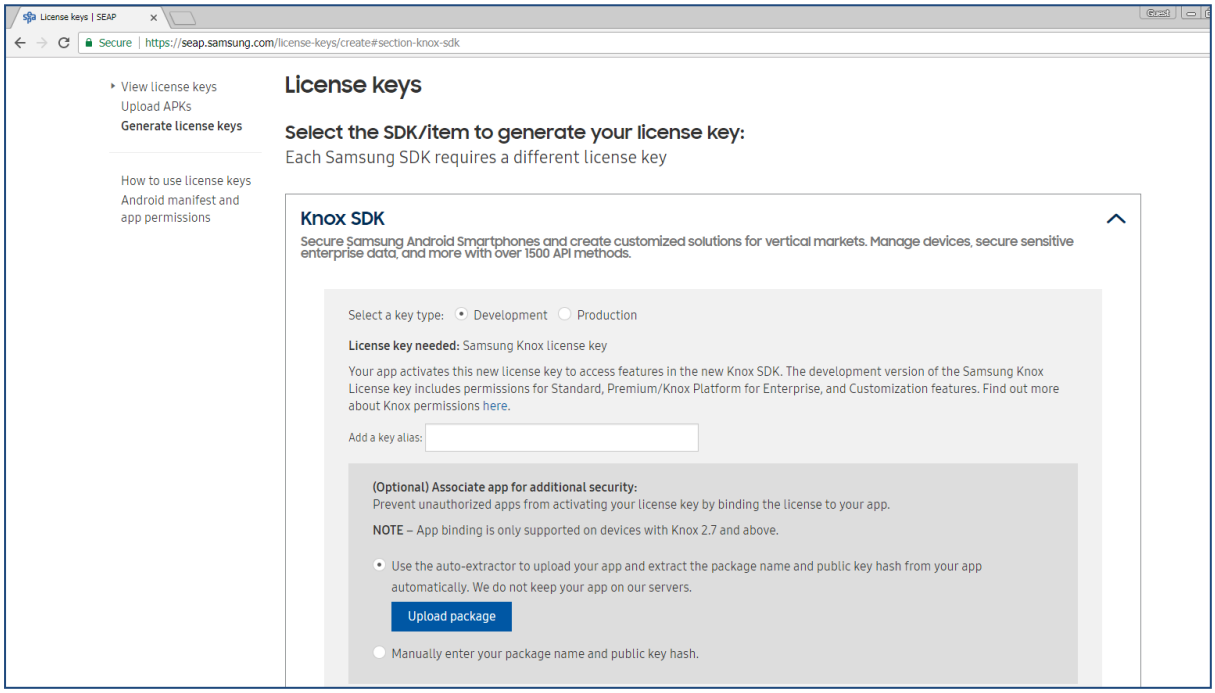


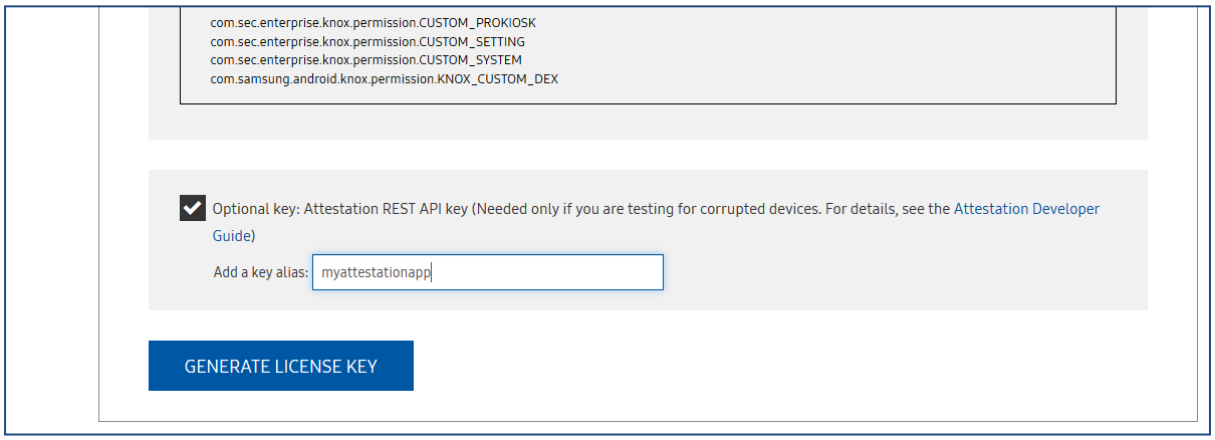
Users can obtain their KNOX licenses on the by open the **Dashboard** page, and then click **View my keys or generated new ones**.



In the license page, click **Generate license key menu**.

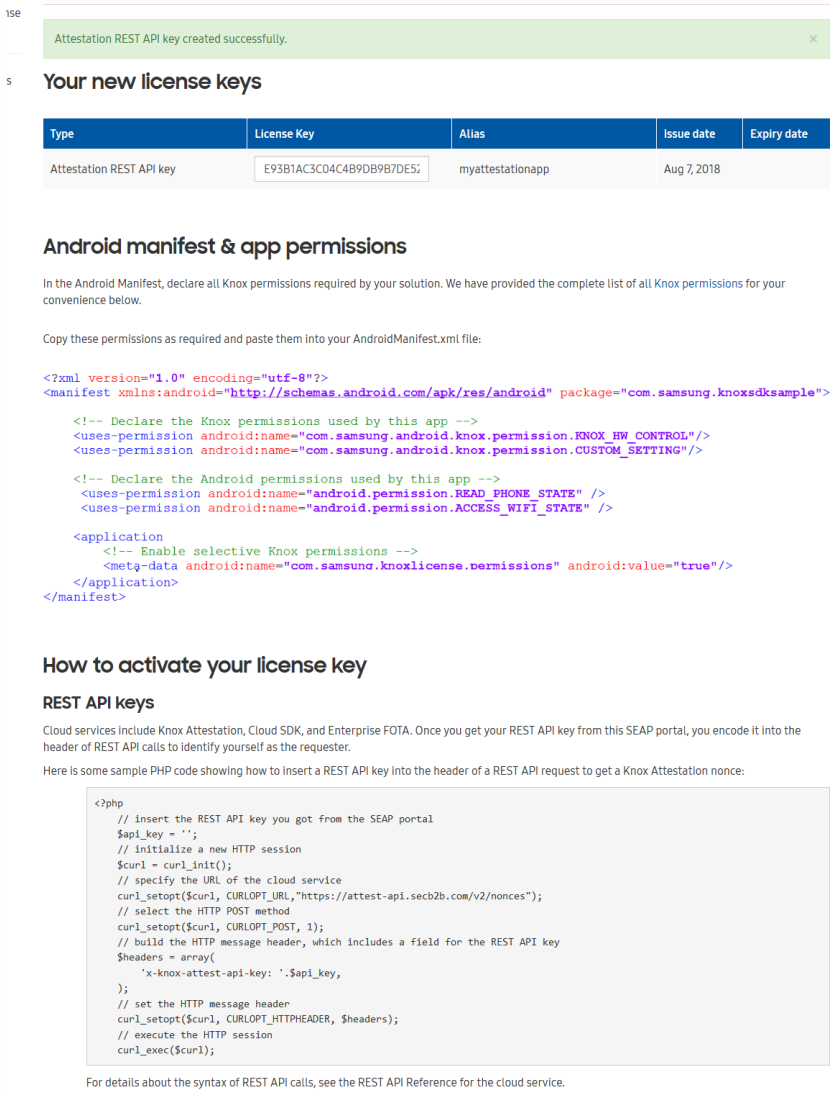






- Select a **Development** key type, which you can use during pre-production on a limited number of devices for a limited time period.
  (Later, when you are ready to commercialize your app, select a Production key.)
- Enter a key alias, for example, MDMKEY.
- For added security, you can identify the app that will use this license; only this app will be allowed to activate this license. During development however, you can skip this step considering your app is not yet ready.
- To use Attestation SDK, you must check the checkbox with label “Optional Key: Attestation REST API Key…” and add key alias.
- Click **GENERATE LICENSE KEY**. The SEAP Not for Resale Agreement appears.

- After you agree to the terms, the Your New License Keys page displays your keys. Keep them secure



Attestation REST API key created successfully.

Your new license keys

Type	License Key	Alias	Issue date	Expiry date
Attestation REST API key	E93B1AC3C04C4B9DB9B7DE5	myattestationapp	Aug 7, 2018	

Android manifest & app permissions

In the Android Manifest, declare all Knox permissions required by your solution. We have provided the complete list of all Knox permissions for your convenience below.

Copy these permissions as required and paste them into your AndroidManifest.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.samsung.knoxsdksample">

    <!-- Declare the Knox permissions used by this app -->
    <uses-permission android:name="com.samsung.android.knox.permission.KNOX_HW_CONTROL"/>
    <uses-permission android:name="com.samsung.android.knox.permission.CUSTOM_SETTING"/>

    <!-- Declare the Android permissions used by this app -->
     <uses-permission android:name="android.permission.READ_PHONE_STATE" />
     <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />

    <application
        <!-- Enable selective Knox permissions -->
        <meta-data android:name="com.samsung.knoxlicense.permissions" android:value="true"/>
    </application>
</manifest>
```

How to activate your license key

REST API keys

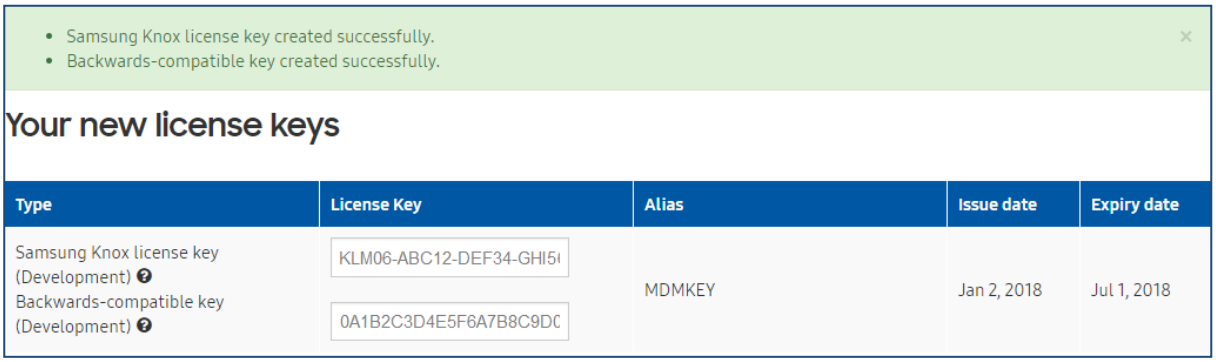
Cloud services include Knox Attestation, Cloud SDK, and Enterprise FOTA. Once you get your REST API key from this SEAP portal, you encode it into the header of REST API calls to identify yourself as the requester.

Here is some sample PHP code showing how to insert a REST API key into the header of a REST API request to get a Knox Attestation nonce:

```
<?php
    // insert the REST API key you got from the SEAP portal
    $api_key = '';
    // initialize a new HTTP session
    $curl = curl_init();
    // specify the URL of the cloud service
    curl_setopt($curl, CURLOPT_URL,"https://attest-ap1.secb2b.com/v2/nonces");
    // select the HTTP POST method
    curl_setopt($curl, CURLOPT_POST, 1);
    // build the HTTP message header, which includes a field for the REST API key
    $headers = array(
        'x-knox-attest-api-key: '.$api_key,
    );
    // set the HTTP message header
    curl_setopt($curl, CURLOPT_HTTPHEADER, $headers);
    // execute the HTTP session
    curl_exec($curl);
```

For details about the syntax of REST API calls, see the REST API Reference for the cloud service.

On the **License Keys** page, users can select **VIEW ALL MY LICENSE KEYS** button to view all license keys linked to users account.



- Samsung Knox license key created successfully.
- Backwards-compatible key created successfully.

Your new license keys

Type	License Key	Alias	Issue date	Expiry date
Samsung Knox license key (Development)	KLM06-ABC12-DEF34-GHI5	MDMKEY	Jan 2, 2018	Jul 1, 2018
Backwards-compatible key (Development)	0A1B2C3D4E5F6A7B8C9D			

Install Prerequisites

To start development using KNOX SDK, some prerequisite applications that need to be installed are:

- Operating System: Microsoft Windows 7/8/10 (32 or 64-bit), Mac OS X 10.8.5 or higher, or Linux 3.2 or higher with GNU C Library (glibc) 2.11
- Java Development Kit (JDK) 8
- Android Studio or Eclipse IDE

Samsung KNOX SDK is compatible with both Android Studio and Eclipse IDE as add-on, however all sample codes in this guide will be presented using Android Studio.

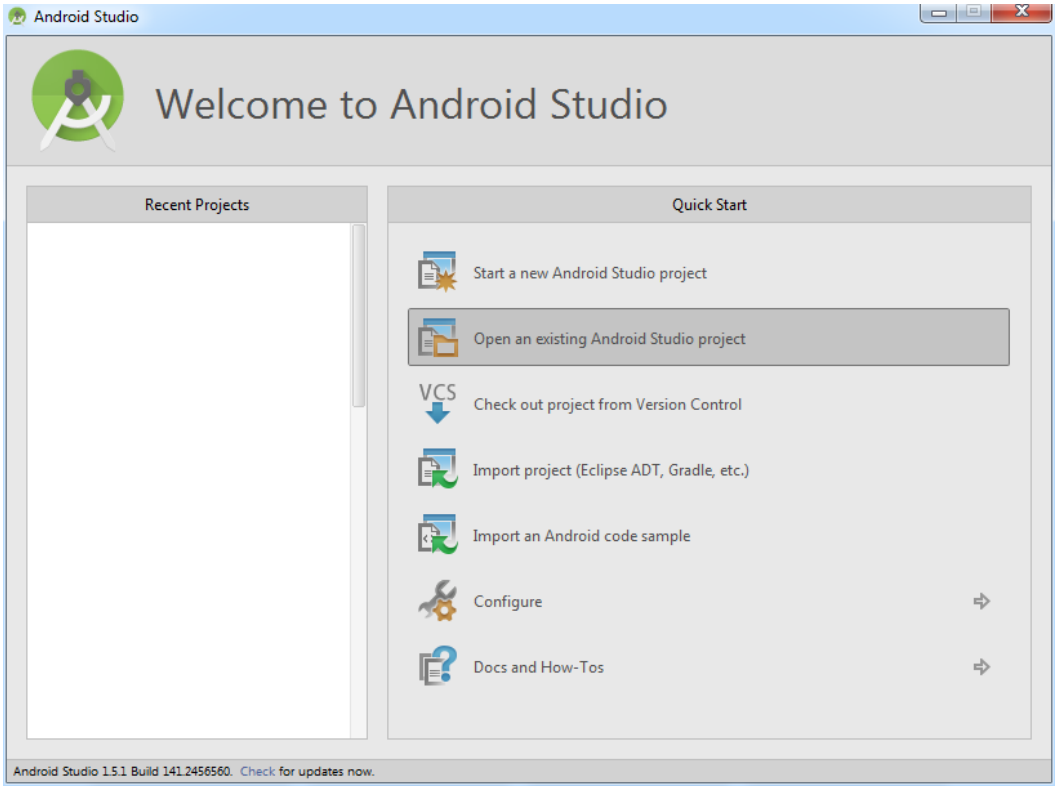
Prior to installing Android Studio, users can open the following link at Android Studio System Requirements to make sure that the system used by users has met the requirement to run Android Studio.

Next, users can start by installing JDK which can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html and continue with installing Android Studio by downloading from http://developer.android.com/sdk/index.html.

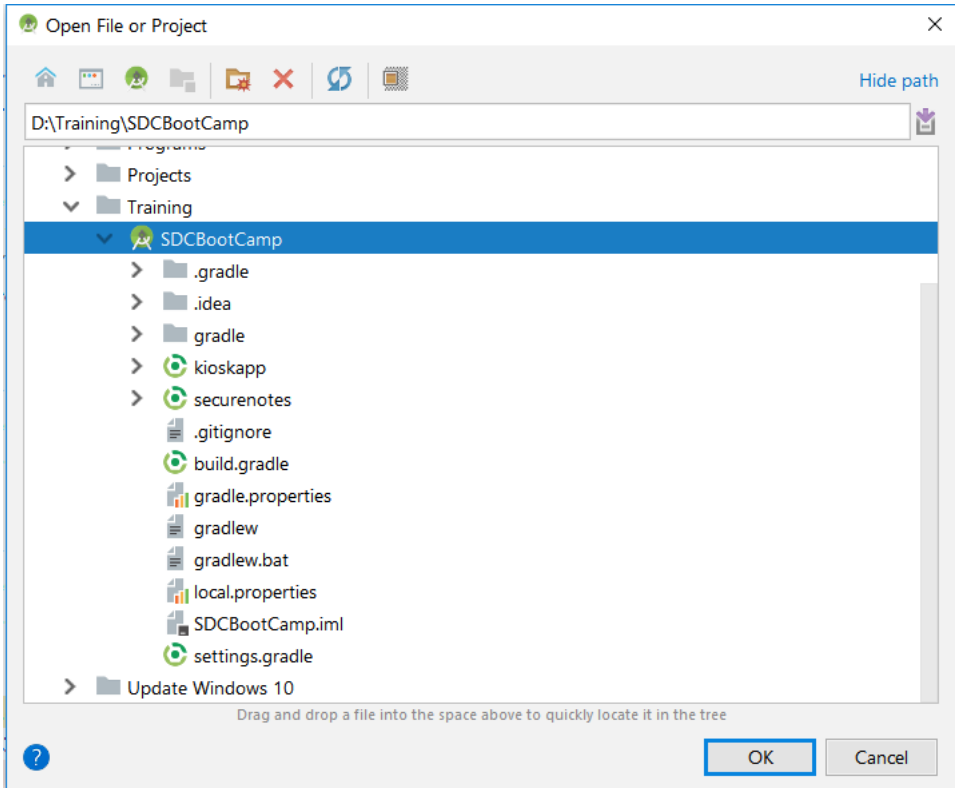
Import Project Template

After Android Studio has been successfully installed on the system, users can now start develop applications using KNOX SDK by creating a new Android Studio project or to use project template provided by this guide that is available at http://bil-id.com/knox-sdk/introduction-template.zip.

Users can import project template to Android Studio by selecting **Open an existing Android Studio project** menu on Android Studio main window.

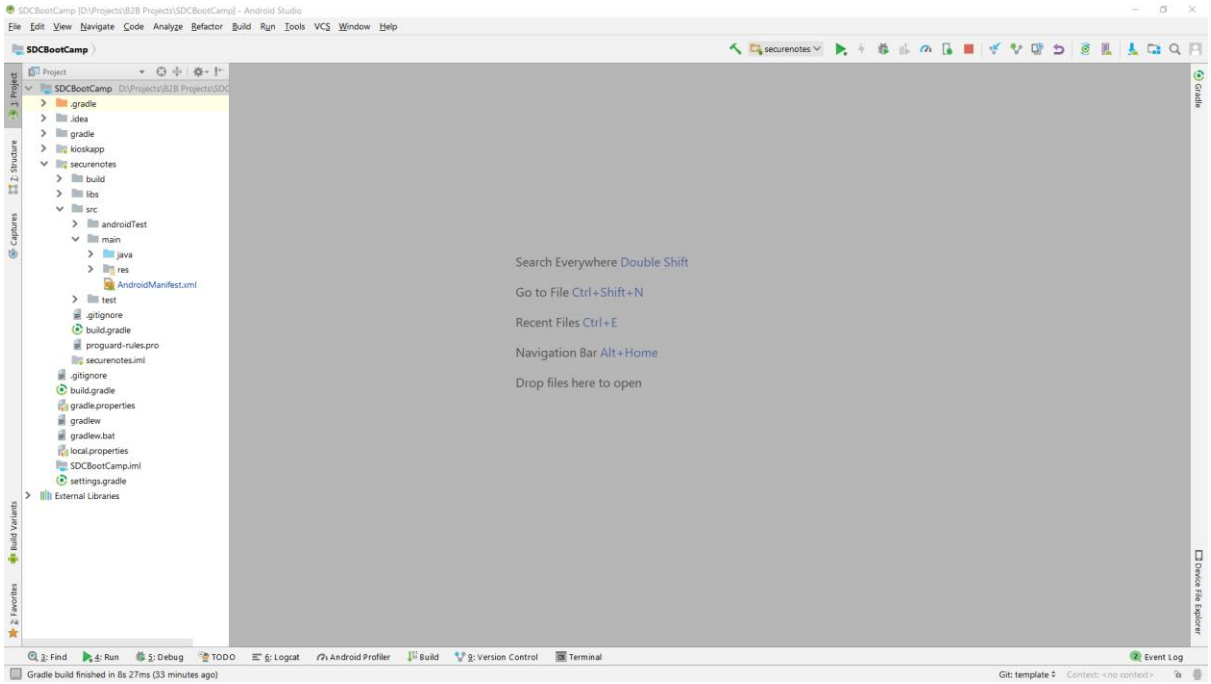


Then select the location of the project template.



Develop with KNOX SDK

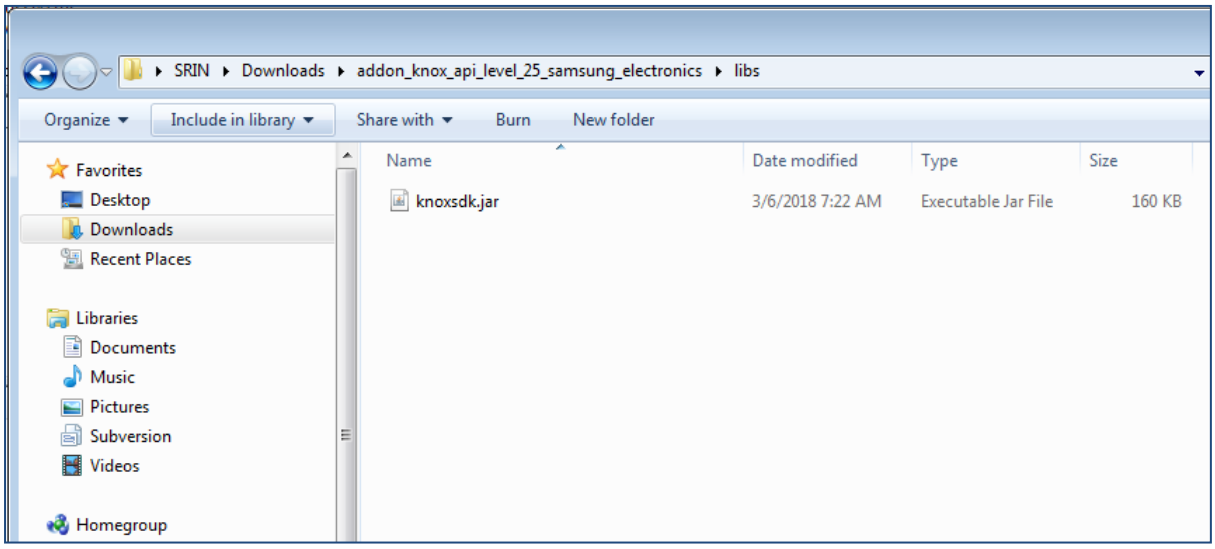
In the project template provided by this guide, there is one Android Studio project available ready to be used by users with only minor changes needed.



Since KNOX SDK is only allowed to be used by users who have been registered as SEAP partner, therefore KNOX SDK libraries will not be included in the project template. Thus users are required to download necessary KNOX SDKs themselves and include them in the project.

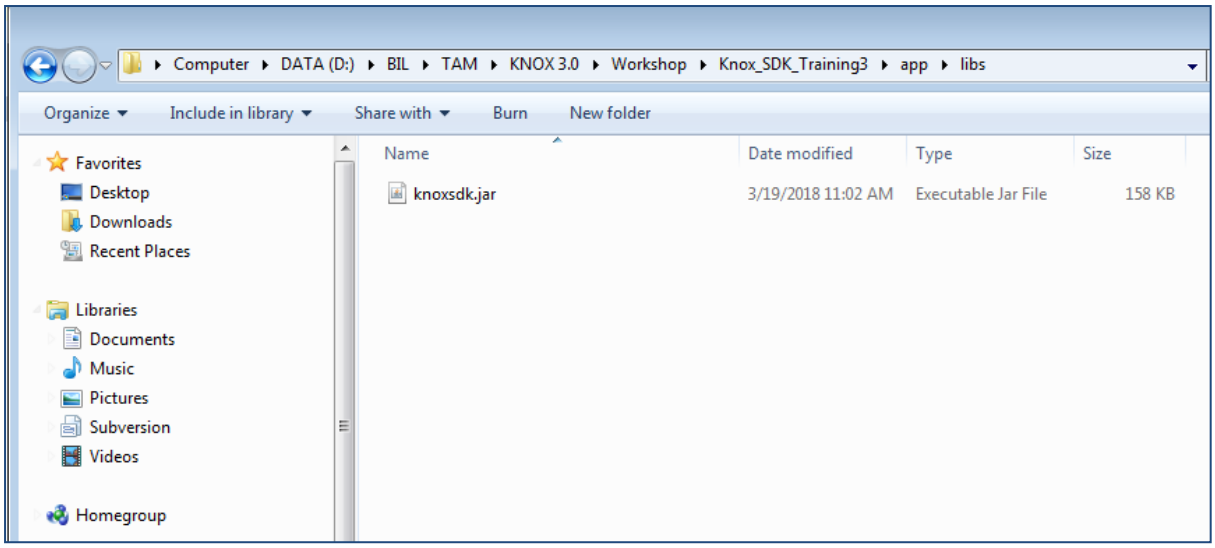
In the downloaded KNOX SDK file there are several jar libraries available for users to be able to use KNOX SDK features in their applications.

For KNOX SDK version 3.0(API level 25), jar library can be found in the **addon_knox_api_level_25_samsung_electronics\libs** directory where it contains a single knoxsdk.jar file:



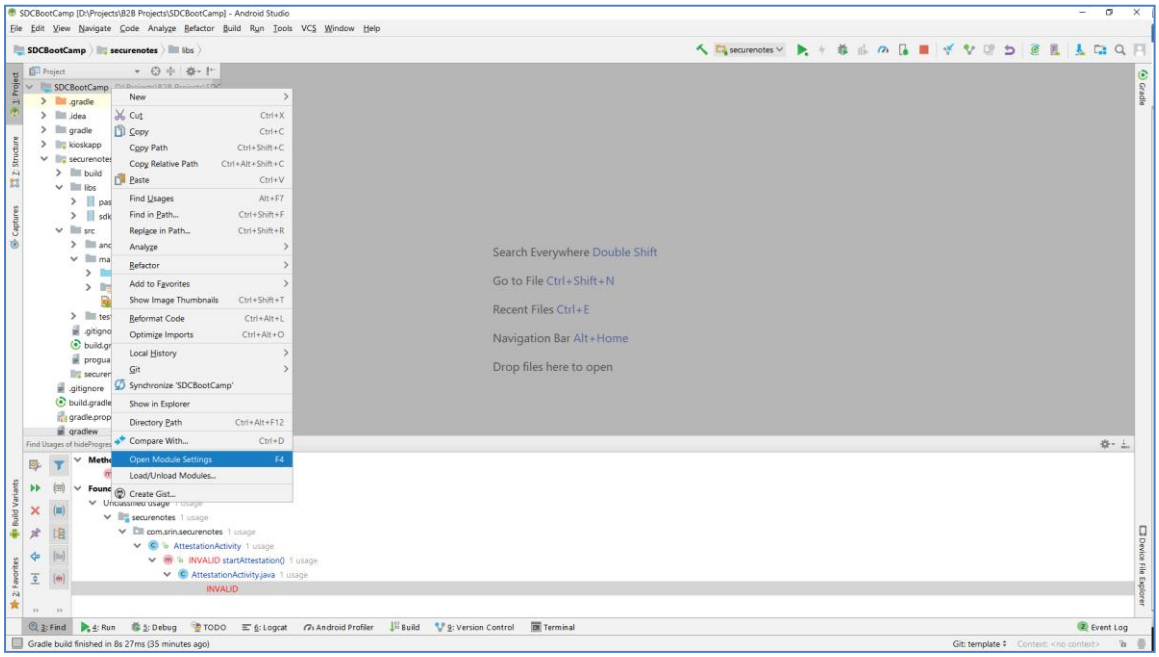
This Introduction to Samsung KNOX SDK guide covers KNOX Standard SDK.

To begin with the courses, users can start by copying the jar library from KNOX SDK file to **app\libs** directory of project template.

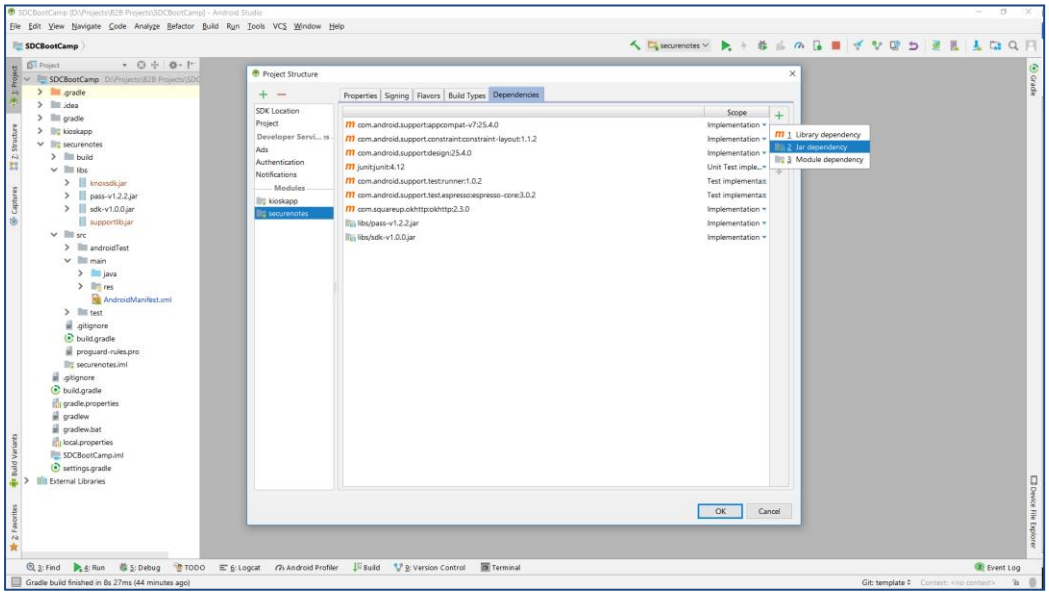


Next, users can include the jar library into the project using these steps.

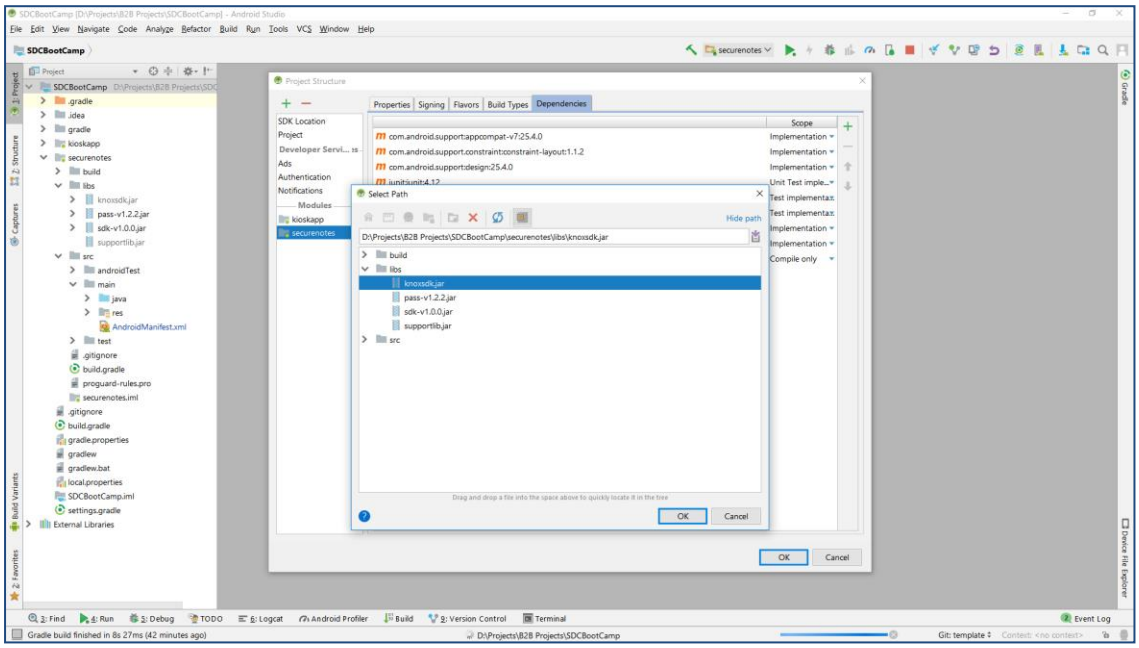
Open module settings window clicking in the **Project** window and choose **Open Module Settings**.



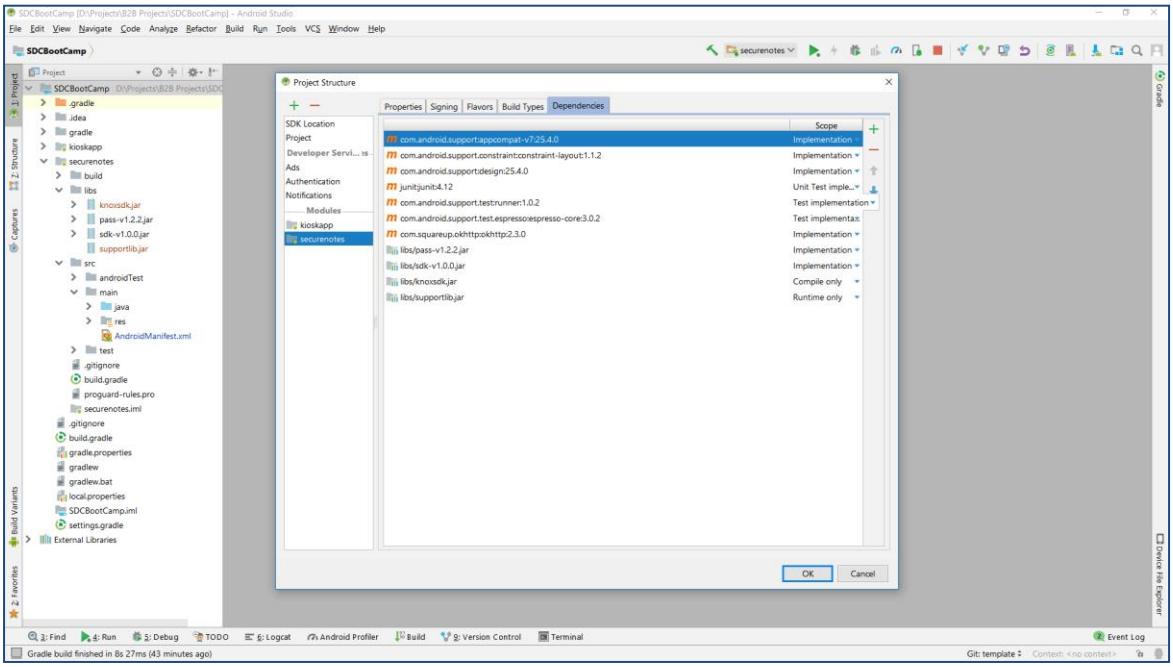
Next, open **Dependencies** settings by selecting **app** module and choose **Dependencies** tab.
Users can then add jar libraries by selecting **+** symbol and choose **File dependency**.



Jar libraries can be added to project one by one using above method.

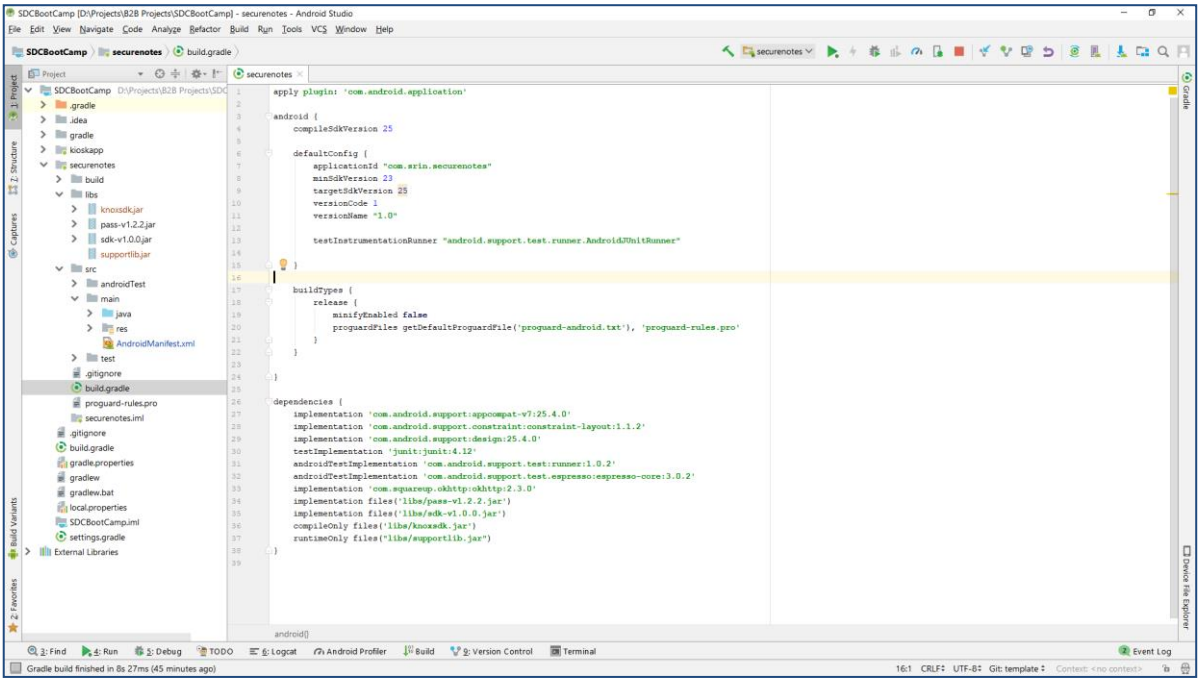


After users have included all the jar libraries into the project, users can now run the applications on the target device.

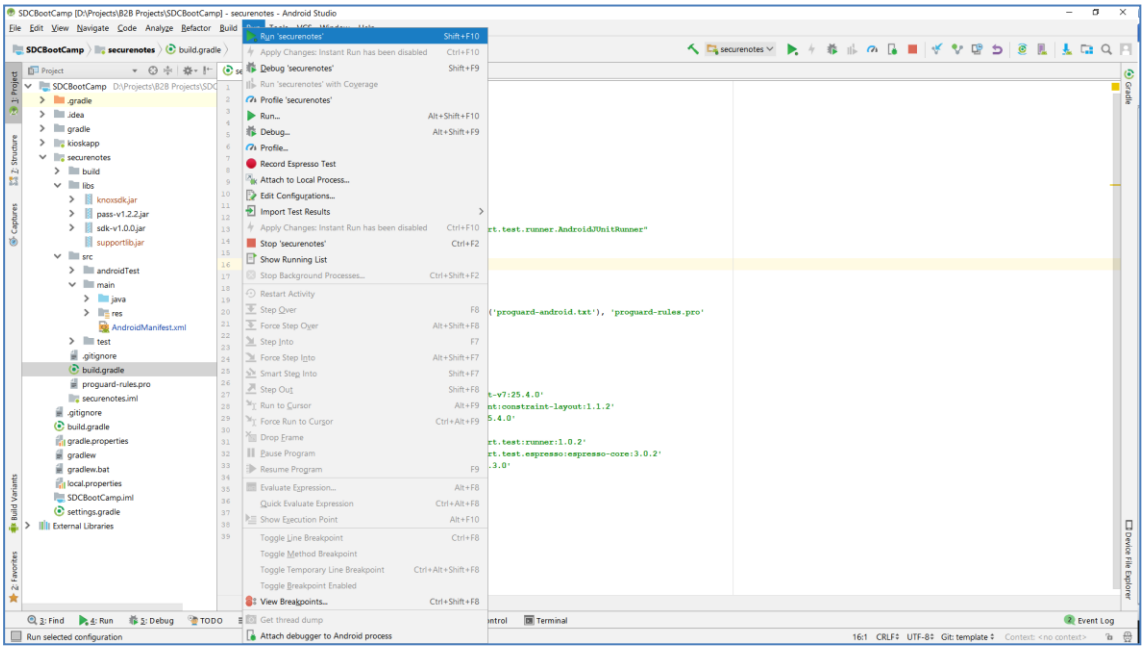


Jar libraries can also be included into the project by inputting these lines of code to the **app\build.gradle** file.

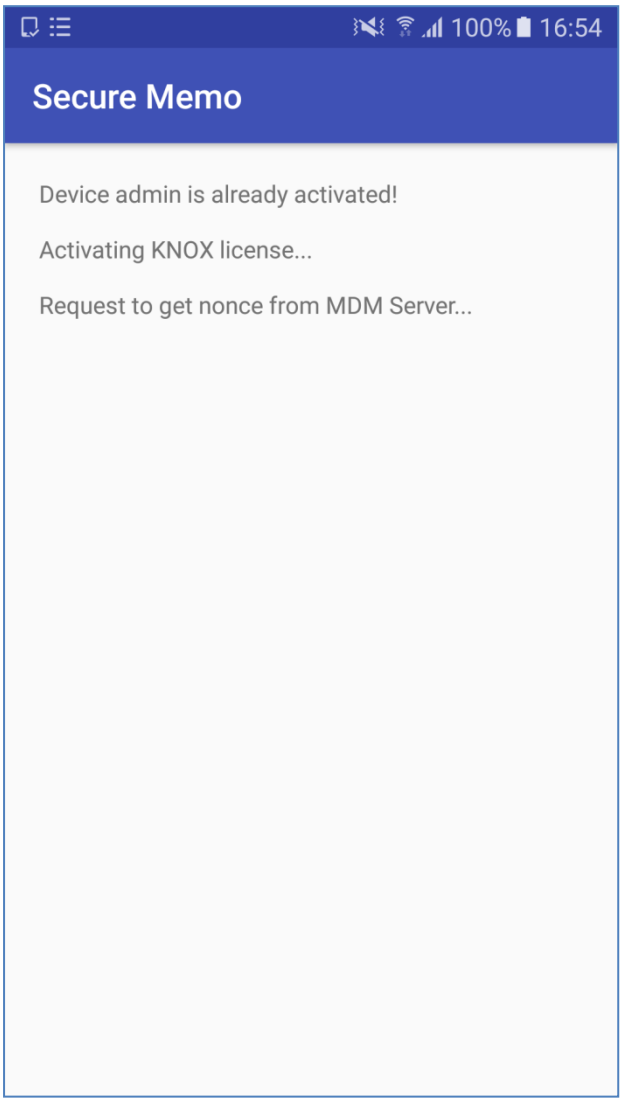
```
compileOnly files('libs/knoxsdk.jar')
runtimeOnly files("libs/supportlib.jar")
```



Lastly, users can run the applications by selecting **Run** menu and choose **Run 'app'**.



Application built from the project template will now run on the device, where basic user interface has been provided to facilitate the Introduction to Samsung KNOX SDK courses to be more focus on implementing KNOX SDK features.



Activate Device Administrator

Every application using KNOX SDK features are required to have device administration features enabled. More details about Android Device Administration API can be found at http://developer.android.com/guide/topics/admin/device-admin.html.

Any function calls to KNOX SDK APIs without having device administration features enabled will throw SecurityException error.

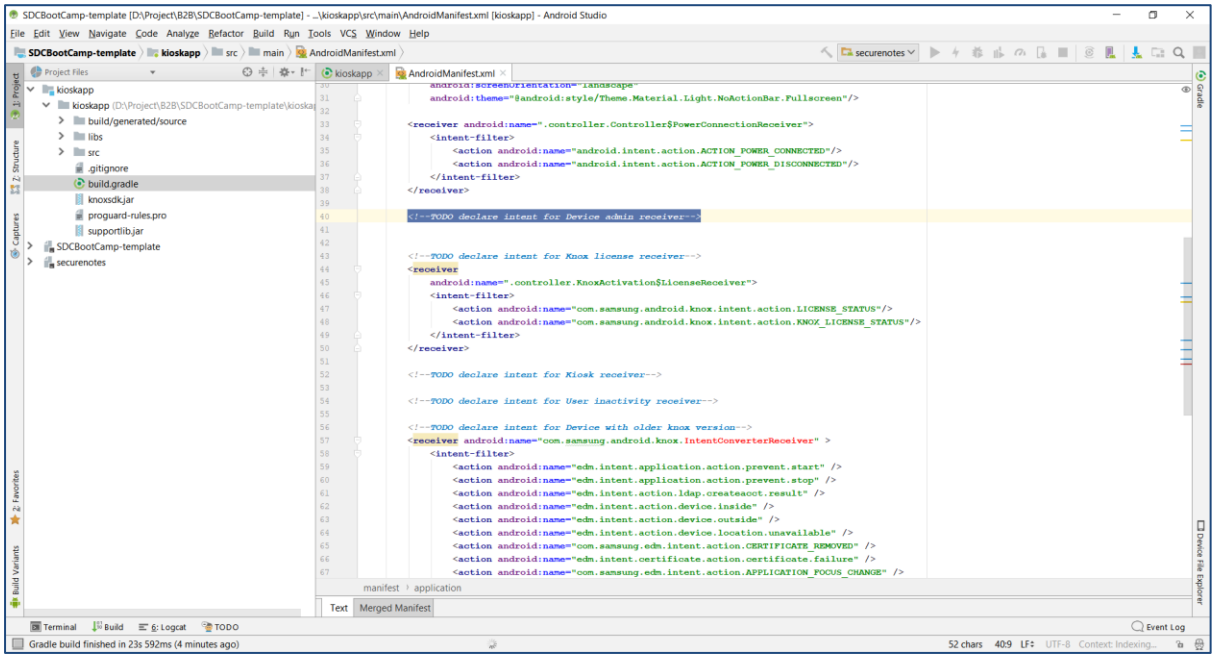
Steps involved in creating device administration applications can be found at http://developer.android.com/guide/topics/admin/device-admin.html#developing or as below

1. Create subclass of DeviceAdminReceiver to application manifest file
2. Include the security policies in metadata file
3. Subclass DeviceAdminReceiver
4. Enable the application as Device Administrator

Create subclass of DeviceAdminReceiver to application manifest file

Applications using Device Administration API have to create a subclass of DeviceAdminReceiver and include it to the application manifest file.

Users who are using the project template can open **app\src\main\AndroidManifest.xml** file and search for **Activate Device Administrator, Step 1** keyword and make necessary changes on the commented block of code.



Code changes in the manifest file can follow code excerpt below.

```
<receiver
    android:name=".controller.KnoxActivation$KnoxAdmin"
    android:description="@string/enterprise_device_admin_description"
    android:label="@string/enterprise_device_admin"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin_receiver"/>
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

Receiver class `.controller.KnoxActivation$KnoxAdmin` which is a subclass of DeviceAdminReceiver should be given `BIND_DEVICE_ADMIN` permission to perform actions in response to `DEVICE_ADMIN_ENABLED` intent broadcasted by system events.

Include the security policies in metadata file

Security policies used by applications should be declared in metadata file which is defined in the manifest file.

In the project template, the metadata file has been provided in **kioskapp\src\main\res\xml\ device_admin_receiver.xml** as defined in the manifest file `android:resource="@xml/enterprise_device_admin"`, and users can fill in the file content as follows.

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
   <uses-policies>
       <encrypted-storage></encrypted-storage>
   </uses-policies>
</device-admin>
```

Users do not need to include all the security policies in metadata file, only the ones that are relevant for the applications.

Failure to include necessary security policies required by KNOX SDK APIs will result to any function calls to those APIs throw SecurityException error.

More details on Android Device Administration API security policy can be found at http://developer.android.com/guide/topics/admin/device-admin.html#policies.

Subclass DeviceAdminReceiver

Next, users have to implement the subclass of DeviceAdminReceiver as included in the manifest file to the code.
This subclass consists of a series of callbacks that are triggered when device administration events occur.

In the project template, this class has been provided in **kioskapp\src\main\java\com\srin\kioskapp\controller\Controller.java** file as an inner class named **KnoxAdmin**.
Users can make necessary changes as the following.

```
public static class KnoxAdmin extends DeviceAdminReceiver {

    @Override
    public void onEnabled(Context context, Intent intent) {
        Toast.makeText(context, "Device Admin enabled",
Toast.LENGTH_SHORT).show();
    }

    @Override
    public CharSequence onDisableRequested(Context context, Intent intent)
{
        return "Disable from Device administrator";
    }

    @Override
    public void onDisabled(Context context, Intent intent) {
        Toast.makeText(context, "Device admin disabled",
Toast.LENGTH_SHORT).show();
    }
}
```

In above code excerpt, only some of callbacks are implemented to KnoxAdmin subclass.
Users can implement other callbacks as can be found at http://developer.android.com/reference/android/app/admin/DeviceAdminReceiver.html.

Enable the application as Device Administrator

The last step is to ask mobile device users to explicitly enable the application as Device Administrator for the security policies declared in metadata file to be enforced.
Mobile device users have to activate device administrator before applications can use any KNOX SDK APIs.

To activate device administrator, users can add the following code excerpt into the project.

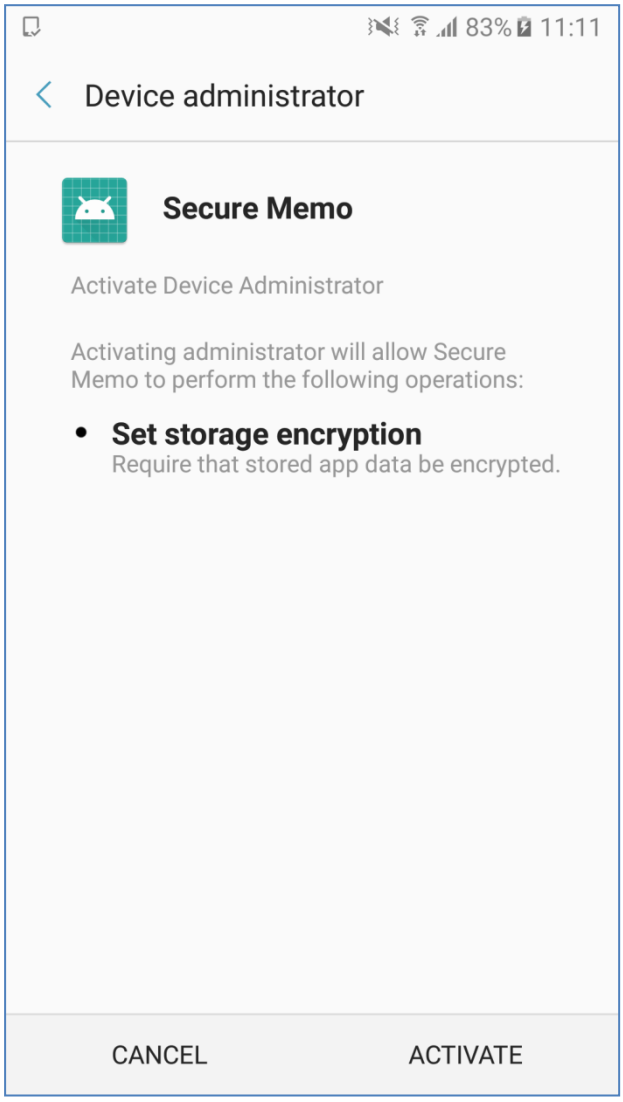
In the project template, there is an empty method named **activateAdmin** to put the code to run this command.

```
public void activateAdmin(Activity activity) {
    if (null == mDPM) {
        Log.e(TAG, "Failed to get DevicePolicyManager");
        return;
    }

    boolean active = mDPM.isAdminActive(mDeviceAdmin);
    if (!active) {
        try {
            Intent intent = new
Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
            intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN,
mDeviceAdmin);
            intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
"Activate device administrator.");
            activity.startActivityForResult(intent, RESULT_ENABLE);
        } catch (Exception e) {
            Log.w(TAG, "Exception: " + e);
        }
    } else {
        Log.w(TAG, "Admin already activated");
    }
}
```

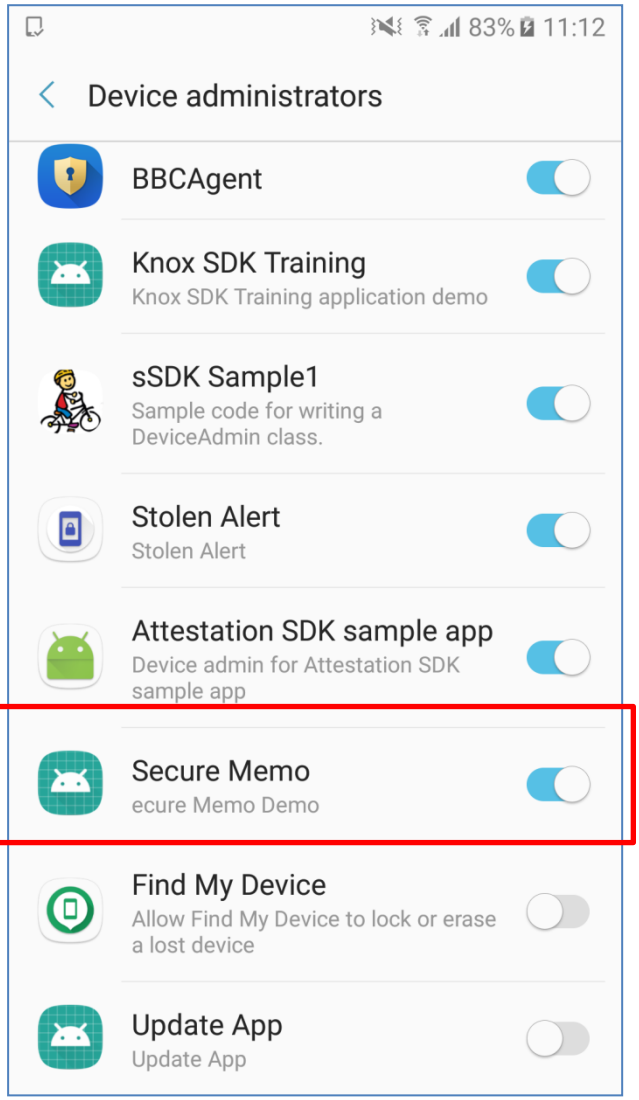
The **mDPM** variable used in above code excerpt is already provided in the project template and initialized as below.

```
mDPM = (DevicePolicyManager)
mContext.getSystemService(Context.DEVICE_POLICY_SERVICE);
```

This command will display a window to prompt users to activate device administrator and enforce security policies as declared in metadata file.



Users can manage device administrator applications by means of **Settings → Lock screen and security → Other security settings → Device administrators**, where mobile device users are allowed to disable device administrator applications added to the system. However developers can also opt to prevent mobile device users to disable device administrator applications using KNOX SDK.



Warning. This guide book does not advise to prevent disable device administrator applications during development, due to any errors in applying KNOX policies by developers might not be reversible and applications may not be able to be uninstalled.

Activate License

Aside from requirement to have device administration enabled, every application using KNOX SDK features are also required to activate KNOX licenses according to KNOX SDKs used by the applications.

Any function calls to KNOX SDK APIs without having KNOX licenses activated for respective SDK will throw SecurityException error.

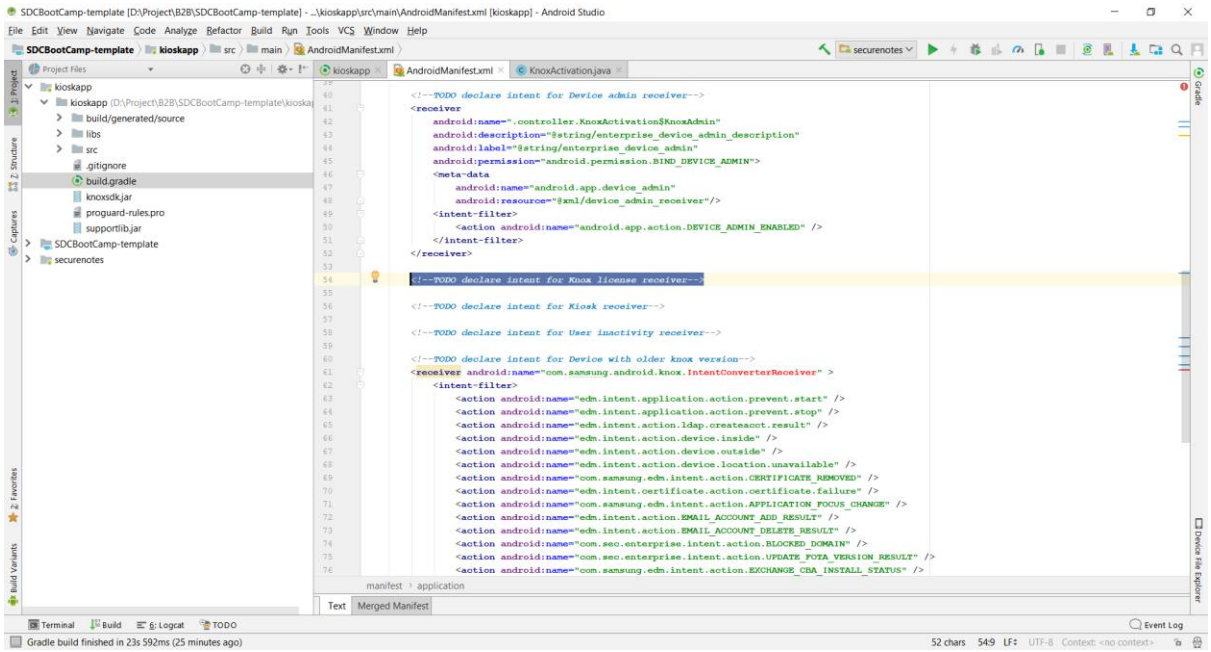
To activate KNOX licenses, users can follow steps as outlined on SEAP portal at these links Samsung KNOX activation APIs and to activate KNOX licenses, or as below.

1. Create subclass of BroadcastReceiver to application manifest file
2. Subclass BroadcastReceiver to respond to KNOX licenses activation events
3. Activate KNOX licenses

Create subclass of BroadcastReceiver to application manifest file

Applications that perform activate KNOX licenses have to create a subclass of BroadcastReceiver and include it to the application manifest file.
This BroadcastReceiver class should be able to respond to
`com.samsung.android.knox.intent.action.LICENSE_STATUS`
intent broadcasted on ELM license activation for Standard SDK.

In the project template, users can open **app\src\main\AndroidManifest.xml** file and search for **Activate License, Step 1** keyword and make necessary changes on the commented block of code.



Code changes in the manifest file can follow code excerpt below.

```
<receiver
   android:name=".controller.KnoxActivation$LicenseReceiver"
   android:enabled="true">
   <intent-filter>
       <action
android:name="com.samsung.android.knox.intent.action.LICENSE_STATUS" />
       <action
android:name="com.samsung.android.knox.intent.action.KNOX_LICENSE_STATUS"
/>
   </intent-filter>
</receiver>
```

Receiver class `.controller.KnoxActivation$LicenseReceiver` which is a subclass of BroadcastReceiver is to respond to KNOX licenses activation events.

Applications using KNOX SDK requires SKL license, required to include `com.samsung.android.knox.intent.action.KNOX_LICENSE_STATUS` in the receiver class. In addition for device with knox version before 2.7.1 requires to include `com.samsung.android.knox.intent.action.LICENSE_STATUS` with backward compatibility key.

Subclass BroadcastReceiver to respond to KNOX licenses activation events

Next, users have to implement the subclass of BroadcastReceiver as included in the manifest file to the code.
This subclass will have the logic to store KNOX licenses activation results to *SharedPreferences* because KNOX SDK has no method to confirm whether applications have activated KNOX licenses.

In the project template, this class has been provided in **kioskapp\src\main\java\com\srin\kioskapp\controller\Controller.java** file as an inner class named **LicenseReceiver**.
Users can search for **LicenseReceiver** keyword and make necessary changes as the following.

```
public static class LicenseReceiver extends BroadcastReceiver {
    private static final String SUCCESS = "success";
    private static final String FAILURE = "fail";

    @Override
    public void onReceive(Context context, Intent intent) {

if(EnterpriseLicenseManager.ACTION_LICENSE_STATUS.equals(intent.getAction())) {
            final String status =
intent.getExtras().getString(EnterpriseLicenseManager.EXTRA_LICENSE_STATUS);
            final int errorCode =
intent.getExtras().getInt(EnterpriseLicenseManager.EXTRA_LICENSE_ERROR_CODE,
Integer.MIN_VALUE);
            if(status.equals(SUCCESS)) {
                SharedPreferences sharedPreferences =
context.getSharedPreferences(LICENSEKNOX, Context.MODE_PRIVATE);
```

```
                SharedPreferences.Editor editor = sharedPreferences.edit();
                editor.putBoolean(ELMKEY, true);
                editor.commit();

                applyPolicy(context);
            } else {
                Log.e(TAG, "ELM Activate license error" +errorCode);
                if(mContext == null) return;
                Toast.makeText(mContext,"ELM Activate license error :
"+errorCode, Toast.LENGTH_SHORT).show();
            }
        }


if(KnoxEnterpriseLicenseManager.ACTION_LICENSE_STATUS.equals(intent.getAction()))
{
            final String status =
intent.getExtras().getString(KnoxEnterpriseLicenseManager.EXTRA_LICENSE_STATUS);
            final int errorCode =
intent.getExtras().getInt(KnoxEnterpriseLicenseManager.EXTRA_LICENSE_ERROR_CODE,
Integer.MIN_VALUE);
            if(status.equals(SUCCESS)) {
                SharedPreferences sharedPreferences =
context.getSharedPreferences(LICENSEKNOX, Context.MODE_PRIVATE);
                SharedPreferences.Editor editor = sharedPreferences.edit();
                editor.putBoolean(SKLKEY, true);
                editor.commit();
                KnoxActivation.getInstance(context).activateELMLicense();
            } else {
                Log.e(TAG, "SKL Activate license error : "+errorCode);
                if(mContext == null) return;
                Toast.makeText(mContext,"SKL Activate license error :
"+errorCode, Toast.LENGTH_SHORT).show();
            }
        }

    }
}
```

To find error code definition sent to receiver class, users can open this link at https://seap.samsung.com/api-references/android-standard/reference/android/app/enterprise/license/EnterpriseLicenseManager.html#ACTION_LICENSE_STATUS for list of ELM license activation error codes.

Activate KNOX licenses

Next, mobile device users have to explicitly accept to terms of privacy policy for applications to be able to use KNOX SDK APIs.

To activate KNOX licenses, users can add the following code excerpt into the project.

In the project template, there are empty method named **activateSKLLicense.** And if want to support devices older than 2.71 user need to **activateELMLicense.** Below is code to run this command.

```
public void activateSKLLicense() {
    if(mDPM.isAdminActive(mDeviceAdmin) && !isSKLLicenseActive(mContext))
{
        mSKL.activateLicense(sklLicense, pkgName);
    }
}
```

```
public void activateELMLicense() {
    if(mDPM.isAdminActive(mDeviceAdmin) && !isELMLicenseActive(mContext))
{
        mELM.activateLicense(elmLicense, pkgName);
    }
}
```

These methods use **mDPM, mSKL** and **mELM** which are already provided in the project template and initialized as below.

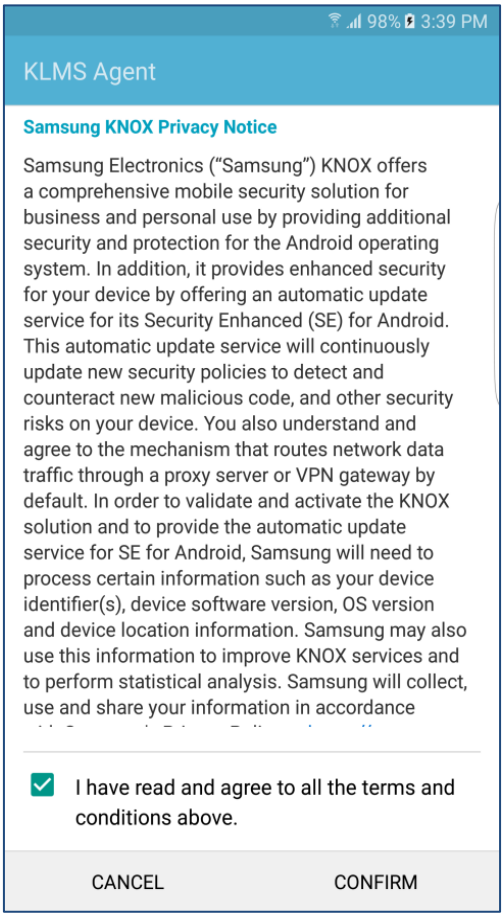
```
mDPM = (DevicePolicyManager)
mContext.getSystemService(Context.DEVICE_POLICY_SERVICE);
mELM = EnterpriseLicenseManager.getInstance(mContext);
mSKL = KnoxEnterpriseLicenseManager.getInstance(mContext);
```

Aside from those, these methods also use **pkgName** variable which is the application package name, and **sklLicense** which is SKL license keys that can be obtained from SEAP portal.

In this example, the app will call **activateSKLLicense** if device admin is already activated or just has been successfully activated.

```
public static class KnoxAdmin extends DeviceAdminReceiver {

    @Override
    public void onEnabled(Context context, Intent intent) {
        KnoxActivation.getInstance(context).activateSKLLicense();
    }
...
public void activateAdmin(Activity activity) {
    ...
    if (!active) {
        ...
    } else {
        Log.w(TAG, "Admin already activated");
        activateSKLLicense();
    }
}
```

This command will display a privacy policy window that needs to be accepted by mobile device users for applications to be able to use KNOX SDKs.



Support older devices

The Knox SDK landing page provides a downloadable library called supportlib.jar, which provides backwards compatibility with older Samsung devices running Knox v2.7 or earlier. These older devices don't recognize the new namespace used by the Knox SDK v3.x.

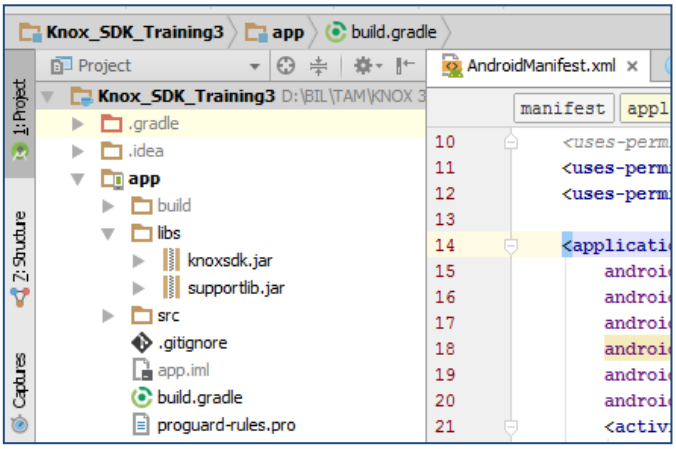
As described in Import the new libraries, you can include the supportlib.jar at run-time, to translate the new namespace used by the Knox SDK v3.x to the old namespace recognized by the older devices.

There are a couple of other things you need to do, which are described in this section:

- Add support library to project
- Define the old intents in the Manifest

Add support library to project

Put the supportlib.jar under the libs folder:



Now add the supportlib.jar to dependencies in by put this following line inside dependencies bracket (app/build.gradle file):

```
dependencies {
    …
    provided files('libs/supportlib.jar')
    apk files('libs/supportlib.jar')
}
```

Define the old intents in the Manifest

Since supportlib.jar is added only for run-time scope and not used in compilation, you need to declare all the old intents in your app's Android manifest file.

Simply copy and paste the following code into your manifest.

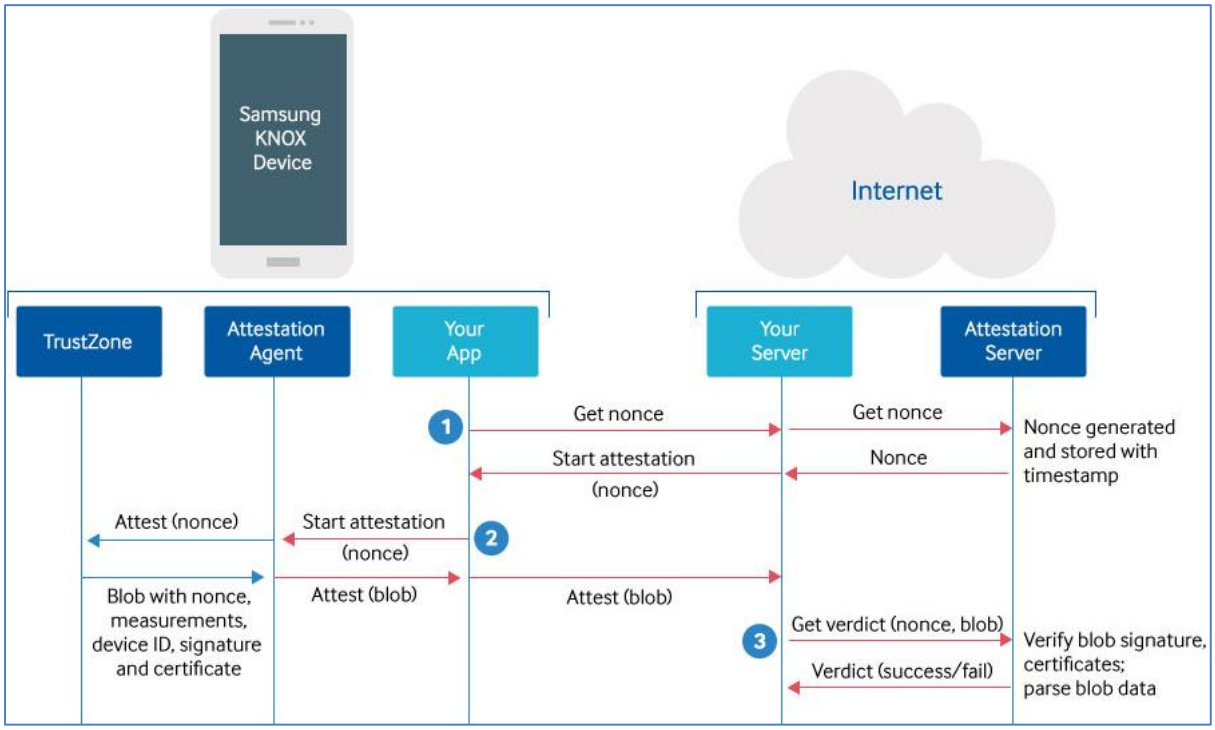
```
<receiver android:name="com.samsung.android.knox.IntentConverterReceiver" >
    <intent-filter>
        <action android:name="edm.intent.application.action.prevent.start" />
        <action android:name="edm.intent.application.action.prevent.stop" />
        <action android:name="edm.intent.action.ldap.createacct.result" />
        <action android:name="edm.intent.action.device.inside" />
        <action android:name="edm.intent.action.device.outside" />
        <action android:name="edm.intent.action.device.location.unavailable" />
        <action android:name="com.samsung.edm.intent.action.CERTIFICATE_REMOVED" />
        <action android:name="edm.intent.certificate.action.certificate.failure" />
        <action android:name="com.samsung.edm.intent.action.APPLICATION_FOCUS_CHANGE" />
        <action android:name="edm.intent.action.EMAIL_ACCOUNT_ADD_RESULT" />
        <action android:name="edm.intent.action.EMAIL_ACCOUNT_DELETE_RESULT" />
        <action android:name="com.sec.enterprise.intent.action.BLOCKED_DOMAIN" />
        <action android:name="com.sec.enterprise.intent.action.UPDATE_FOTA_VERSION_RESULT" />
        <action android:name="com.samsung.edm.intent.action.EXCHANGE_CBA_INSTALL_STATUS" />
        <action android:name="android.intent.action.sec.CBA_INSTALL_STATUS" />
        <action android:name="edm.intent.action.EXCHANGE_ACCOUNT_ADD_RESULT" />
        <action android:name="edm.intent.action.EXCHANGE_ACCOUNT_DELETE_RESULT" />
        <action android:name="com.samsung.edm.intent.action.ENFORCE_SMIME_ALIAS_RESULT" />
        <action android:name="edm.intent.action.knox_license.status" />
        <action android:name="edm.intent.action.license.status" />
        <action android:name="com.samsung.edm.intent.event.NTP_SERVER_UNREACHABLE" />
        <action android:name="edm.intent.action.enable.kiosk.mode.result" />
        <action android:name="edm.intent.action.disable.kiosk.mode.result" />
        <action android:name="edm.intent.action.unexpected.kiosk.behavior" />
        <action android:name="com.samsung.edm.intent.action.SIM_CARD_CHANGED" />
        <action android:name="android.intent.action.sec.SIM_CARD_CHANGED" />
        <action android:name="com.samsung.action.knox.certenroll.CEP_CERT_ENROLL_STATUS" />
        <action android:name="com.samsung.action.knox.certenroll.CEP_SERVICE_DISCONNECTED" />
        <action android:name="com.sec.enterprise.knox.intent.action.KNOX_ATTESTATION_RESULT" />
        <action android:name="com.sec.action.NO_USER_ACTIVITY" />
        <action android:name="com.sec.action.USER_ACTIVITY" />
        <action android:name="com.samsung.android.mdm.VPN_BIND_RESULT" />
    </intent-filter>
</receiver>
```

Attestation

AttestationPolicy API allows to verify the integrity of a device. You can check if a device has been rooted or is running unauthorized firmware.

To perform a reliable attestation check, you must create both an Android app to initiate the attestation check on a device as well as a web script to communicate with Samsung's Attestation server.

Here is the end-to-end process:



Get Nonce

A nonce is a random value that uniquely identifies each attestation request. Each nonce is valid for a short time period. This is for security purposes, so that the Attestation Server can fail any request made using that nonce after the valid period. This avoids replay attacks that could allow an attacker to access a past attestation result.

In your Android app, request a nonce from your web server. On the web server, you need a script to take the request and forward it to Samsung's Attestation server. Code for the web server can be seen on SEAP portal.

This application is using *okhttp* library for HTTP request. Each HTTP request will be wrapped inside *AsyncTask,* so as not to block other processes. A *Listener* will be used for AsyncTask, so that the result can be observed by other method. The first is to request Nonce from server. To do that, call function `HttpClient.getInstance().getNonce(..,..)` inside `doInBackground` of `GetNonceTask`.

```
// AttestationActivity.java
private static final String URL_MDM_SERVER_NONCE = "http://bil-
id.com/attestation/nonces";
private static final String URL_MDM_SERVER_MEASUREMENT = "http://bil-
id.com/attestation/measurements";
…
public static class GetNonceTask extends AsyncTask<String, String, String> {
    private NonceTaskListener listener;

    @Override
    protected String doInBackground(String... strings) {
        String response = null;
        try {
            response = HttpClient.getInstance()
                    .getNonce(URL_MDM_SERVER_NONCE, PolicyController.attestationAPIKey);
        } catch (IOException e) {
            //Handle IOException when get nonce response from MDM server
            e.printStackTrace();
        }
        return response;
    }

    @Override
    protected void onPostExecute(String s) {
        listener.onNonceTaskPostkExecute(s);
    }

    @Override
    protected void onProgressUpdate(String... values) {
        listener.onNonceTaskProgressUpdate(values[0]);
    }

    public void setListener(NonceTaskListener listener) {
        this.listener = listener;
    }

    public interface NonceTaskListener {
        void onNonceTaskPostkExecute(String s);
        void onNonceTaskProgressUpdate(String values);
    }
}
…
```

Pass Attestation API key you have got from SEAP into request header inside each HTTP request. You can see the request implementation on *remote/HttpClient.java* class.

```
// remote/HttpClient.java
public class HttpClient {
…
public String getNonce(String url, String apiKey) throws IOException {
    String post = "";
    RequestBody body = RequestBody.create(JSON, post);
    Request request = new Request.Builder()
            .url(url)
            .post(body)
            .addHeader("x-knox-attest-api-key", apiKey)
            .addHeader("Accept", "application/json")
            .build();
```

```
    Response response = mOk.newCall(request).execute();
    return response.body().string();
}

...
}
```

Server will give response in a JSON object which contains the nonce. The response is passed to the listener that has been set inside `GetNonceTask`. You can set the implementation of the listener by calling `getNonceTask.setListener` before executing the `getNonceTask.`

```
// AttestationActivity.java
getNonceTask = new GetNonceTask();
getNonceTask.setListener(new GetNonceTask.NonceTaskListener() {
    @Override
    public void onNonceTaskPostkExecute(String response) {
        if (response != null && !response.isEmpty()) {
            JSONObject jsonObject = null;
            try {
                jsonObject = new JSONObject(response);
                mNonce = jsonObject.getString("nonce");
                updateProgressStatus("Get nonce: " + mNonce);

PolicyController.getInstance(AttestationActivity.this).startAttestation(mNonce);
            } catch (JSONException e) {
                //Handle JSONException when get nonce from response
                e.printStackTrace();
            }
        }
    }

    @Override
    public void onNonceTaskProgressUpdate(String values) {
        updateProgressStatus(values);
    }
});

getNonceTask.execute(null, null);
```

Start Attestation

When your Android app gets a nonce from your web server, send it to Samsung's Attestation agent on the device and set up intent to handle the response.

When the Knox Attestation agent has a blob containing the attestation measurements, it emits the intent **com.samsung.android.knox.intent.action.KNOX_ATTESTATION_RESULT**.

Identify the class that handles this intent. In our sample app com.samsung.business.sdk.attestation, we use the class AttestationReceiver. Declare the *receiver* in the manifest file as follows:

```
// AndroidManifest.xml
<receiver
    android:name=".controller.PolicyController$AttestationReceiver"
    android:enabled="true">
    <intent-filter>
        <action
android:name="com.samsung.android.knox.intent.action.KNOX_ATTESTATION_RESULT" />
    </intent-filter>
</receiver>
```

Your app handles the intent `com.sec.enterprise.knox.intent.action.KNOX_ATTESTATION_RESULT` by passing it to `AttestationReceiver.onReceive()`. This method is where you handle the blob sent by the Knox Attestation Agent.

```
// PolicyController.java
public static class AttestationReceiver extends BroadcastReceiver {
    private static final String TAG = "Attestation";

    static AttestationActivity activity;

    public AttestationReceiver() { }

    // Get the current fragment instance
    public AttestationReceiver(AttestationActivity activity) {
        AttestationReceiver.activity = activity;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        if (AttestationPolicy.ACTION_KNOX_ATTESTATION_RESULT.equals(intent
                .getAction())) {

            final int result = intent.getIntExtra(AttestationPolicy.EXTRA_RESULT,
Integer.MIN_VALUE);
            final String errorMsg = intent.getExtras().getString(
                    AttestationPolicy.EXTRA_ERROR_MSG);

            switch (result) {
                //The communication with the TrustZone and the Attestation server was
successful
                case AttestationPolicy.ERROR_NONE:
                    byte[] blob =
intent.getByteArrayExtra(AttestationPolicy.EXTRA_ATTESTATION_DATA);
                    activity.updateProgressStatus(
                            "The communication with the TrustZone and the Attestation
server was successful");
                    activity
                            .updateProgressStatus("Get blob from attestation
complete...");
                    activity.requestAttestationStatus(mNonce, blob);
                    activity
                            .updateProgressStatus("Start get attestation status
process...");
                    break;
                //Device is not compatible for attestation
                case AttestationPolicy.ERROR_DEVICE_NOT_SUPPORTED:
                    if (errorMsg != null) {
                        Log.w(TAG, errorMsg);
                    }
                    activity
```

```
                            .updateProgressStatus("Device is not compatible for
attestation");
                    break;
                //invalid nonce provided.
                case AttestationPolicy.ERROR_INVALID_NONCE:
                    activity.updateProgressStatus("Invalid nonce provided");
                    break;
                //Only MDM partner client with appropriate permission can invoke this
API
                case AttestationPolicy.ERROR_MDM_PERMISSION:
                    activity.updateProgressStatus(
                            "Only MDM partner client with appropriate permission can
invoke this API");
                    break;
                //Error communicating with the trust zone
                case AttestationPolicy.ERROR_TIMA_INTERNAL:
                    activity
                            .updateProgressStatus("Error communicating with the trust
zone");
                    break;
                //Unknown error during attestation process
                case AttestationPolicy.ERROR_UNKNOWN:
                    if (errorMsg != null) {
                        Log.w(TAG, errorMsg);
                    }
                    activity
                            .updateProgressStatus("Unknown error during attestation
process");
                    break;

                default:
                    activity
                            .updateProgressStatus("Unknown error during attestation
process");
                    break;
            }
        }
    }
}
```

After the *receiver* has been set, you can start attestation by using the nonce that you got from `getNonceTask` and call AttestationPolicy.startAttestation(String nonce).

```
// PolicyController.java
public void startAttestation (String nonce) {
    mNonce = nonce;
    AttestationPolicy attestation = new AttestationPolicy(mContext);
    try {
        attestation.startAttestation(nonce);
    } catch (SecurityException e) {
        Log.w(TAG, "SecurityException: " + e);
    }
}
```

Get Attestation Verdict

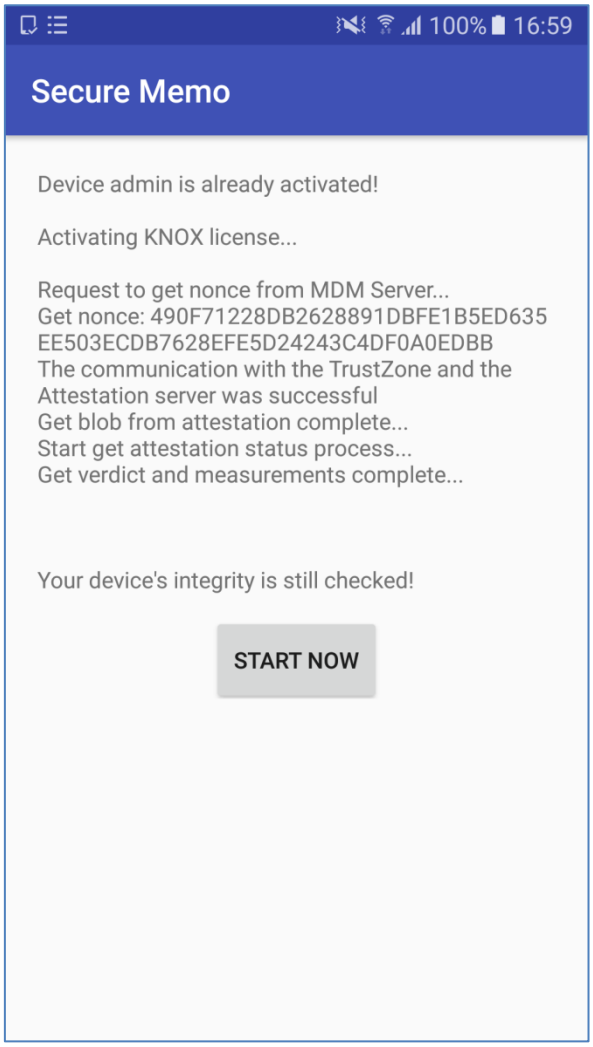
Now that you have a blob containing the integrity measurements (in `AttestationReceiver`), send it to your web server, which in turn sends it to Samsung's Attestation server to get a verdict on whether or not the device passed the integrity checks.
Send the blob to your web server using the HTTP protocol. On the web server, you need a script to take the blob and forward it to Samsung's Attestation server. Web server will result Attestation verdict which you will parse the response into a JSON object. Call `HttpClient.getInstance().getAttestationStatus(url, blob)` inside `GetMeasurementTask` to get the verdict from attestation server.

```
// AttestationActivity.java
public static class GetMeasurementTask extends AsyncTask<Data, String, String> {

    private MeasurementTaskListener listener;

    @Override
    protected String doInBackground(Data... data) {
        String response = null;
        try {
            response = HttpClient.getInstance()
                    .getAttestationStatus(URL_MDM_SERVER_MEASUREMENT + "?nonce=" +
data[0].nonce,
                            data[0].blob, PolicyController.attestationAPIKey);
        } catch (IOException e) {
            //Handle IOException when get attestation status response from MDM server
            e.printStackTrace();
        }
        return response;
    }

    @Override
    protected void onPostExecute(String s) {
        listener.onMeasurementTaskPostkExecute(s);
    }

    @Override
    protected void onProgressUpdate(String... values) {
        listener.onMeasurementTaskProgressUpdate(values[0]);
    }

    public void setListener(MeasurementTaskListener listener) {
        this.listener = listener;
    }

    public interface MeasurementTaskListener {
        void onMeasurementTaskPostkExecute(String s);
        void onMeasurementTaskProgressUpdate(String values);
    }
}
…
```

Check verdict (a string which could be: "Yes", "No", "Unknown") result from web server via the *listener*.

```
// AttestationActivity.java
public void requestAttestationStatus(final String nonce, final byte[] blob)
{
    String response;
    //Use AsyncTask to prevent blocking when calling getAttestationStatus
using synchronous HTTP request
    getMeasurementTask = new GetMeasurementTask();
    getMeasurementTask.setListener(new
GetMeasurementTask.MeasurementTaskListener() {
        @Override
        public void onMeasurementTaskPostkExecute(String response) {
            if (!response.isEmpty()) {
                mAttestationStatus = response;
                updateProgressStatus("Get verdict and measurements
complete...\n\n");
                if (!mAttestationStatus.isEmpty()) {
                    JSONObject jsonObject = null;
                    StringBuffer buff = new StringBuffer();
                    try {
                        jsonObject = new JSONObject(mAttestationStatus);
                        String verdict = jsonObject.getString("verdict");
                        if (verdict != null && verdict.equals("Yes")) {
                            buff.append("\n");
                            buff.append("Your device's integrity is still
checked!");

mBtnStartApplication.setVisibility(View.VISIBLE);
                        } else {
                            buff.append("\n");
                            buff.append("Your device may have been
tampered!");

mBtnStartApplication.setVisibility(View.VISIBLE);
                        }

                    } catch (JSONException e) {
                        //Handle JSONException when get status from
response
                        e.printStackTrace();
                    }

                    updateProgressStatus(buff.toString());
                }
            }
        }

        @Override
        public void onMeasurementTaskProgressUpdate(String values) {
            updateProgressStatus(values);
        }
    });

    Data data = new Data();
    data.nonce = nonce;
```

```
    data.blob = blob;
    getMeasurementTask.execute(data);
}
```



Secure Data Protection (SDP)

Sensitive Data Protection (SDP) is used to keep confidential data encrypted, even after a device boots up.
Samsung Knox provides defense-grade Sensitive Data Protection (SDP) that is specifically designed to meet the Mobile Device Fundamentals Protection Profile (MDFPP) requirements defined by the National Information Assurance Partnership (NIAP) for DAR.Through SDP, you can protect sensitive data both inside and outside a Knox container.

Even if a device is rooted, Knox can guard the integrity of your data. When it detects that an unofficial bootloader or firmware file is installed on the device, Knox permanently fuses a Knox Warranty Bit on the device, and as a result destroys the encryption key for the Knox file system.

Through Knox SDP, you can:

- **Protect sensitive databases** — You can apply sensitive data protection to selected databases and database columns.
- **Protect sensitive files** — You can protect an app's files with sensitive data protection.
- **Create a custom SDP engine** — By default, you use a default SDP engine, which uses device unlock credentials to generate cryptographic keys for encryption. Alternatively, you can create your own custom SDP engine, by providing an app-specific password (which could be provided by the user) to generate cryptographic keys.

Add SDP Meta Tag

To enable SDB, add this meta data on *AndroidManifest*.

```
// AndroidManifest.xml
<meta-data
    android:name="sdp"
    android:value="enabled" />
```

Create SDP Engine

This app will use custom SDP engine where you can set app-specific password to generate the cryptographic key. It is up to the app to set or reset the password that is used.

```
// MainActivity.java

private void createSDPEngine() {
    try {
        if(SdpEngine.getInstance().exists(SDP_ALIAS)) return;
        SdpCreationParamBuilder builder = new SdpCreationParamBuilder(SDP_ALIAS,
                SdpEngineConstants.Flags.SDP_MINOR);
        builder.addPrivilegedApp(new SdpDomain(SDP_ALIAS, getPackageName()));
        SdpEngine.getInstance().addEngine(builder.getParam(), SDP_PASSWORD, null);
        SdpEngine.getInstance().lock(SDP_ALIAS);
    } catch (SdpException e) {
        e.printStackTrace();
    }
}
```

Write File into SDP

To write file into SDP, you must get SDP file system. But, before you do that, unlock the SdpEngine by calling SdpEngine.*getInstance*().unlock(***alias, password***). After that, You can access SDP file system by calling getFilesDir() from SdpFileSystem object. *SdpFileSystem.getFilesDir()* will return directory root in the encrypted filesystem. Then you can use File object to save the text. Use setSensitive(file) to mark a file as sensitive. A sensitive file means the file will only be decrypted only on Sdp is unlock using *unlock(alias, password)* method.

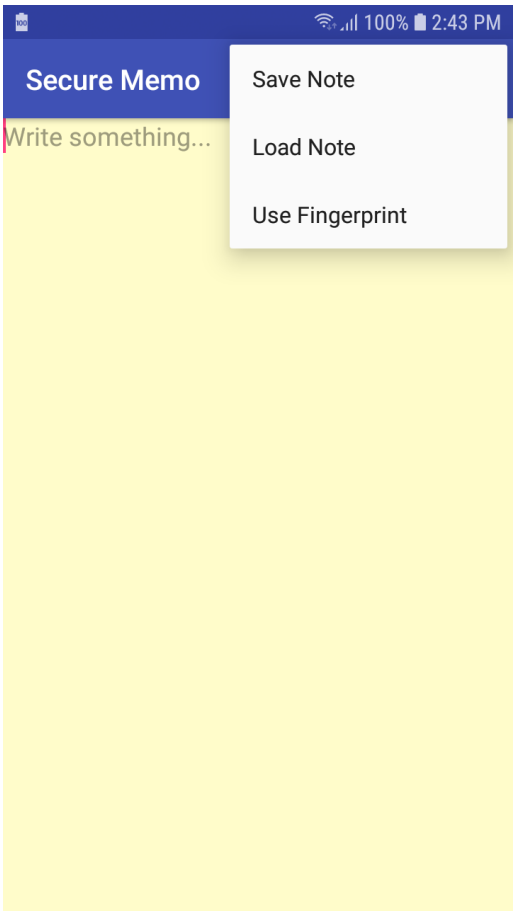
```
// MainActivity.java
private void saveFile() {
     try {
    String text = editTextView.getText().toString();
    if (!text.isEmpty()) {

        SdpEngine.getInstance().unlock(SDP_ALIAS, SDP_PASSWORD);

        SdpFileSystem sdpFileSystem = new SdpFileSystem(this, SDP_ALIAS);
        File file = new File(sdpFileSystem.getFilesDir(), "note.txt");

        FileOutputStream outputStream = new FileOutputStream(file);
        outputStream.write(editTextView.getText().toString().getBytes());
        outputStream.close();

        boolean result;
        if(!sdpFileSystem.isSensitive(file)) {
            result = sdpFileSystem.setSensitive(file);
            if (result) {
                Toast.makeText(this, "Success save to '" + file.getAbsolutePath() + "'",
```

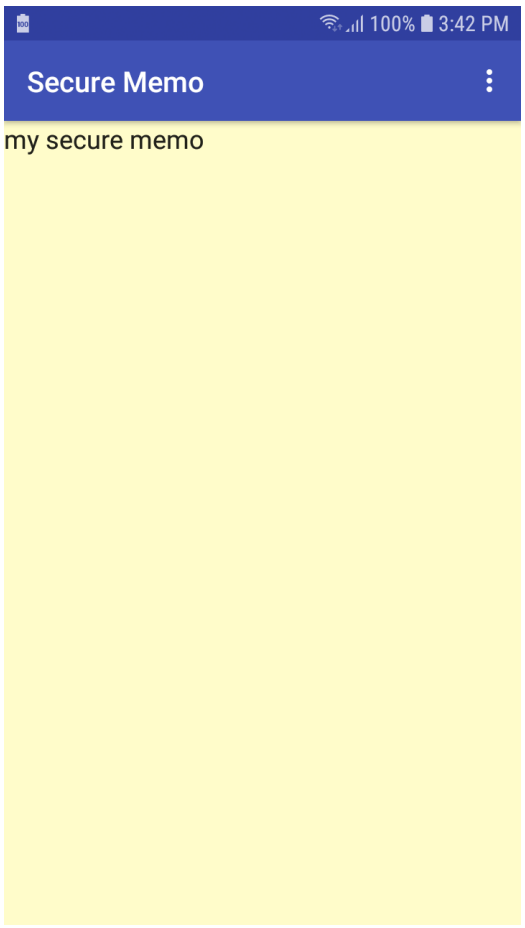
```
Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(this, "Failed save to SDP!", Toast.LENGTH_SHORT).show();
            }

            return;
        }

        Toast.makeText(this, "Success save to '" + file.getAbsolutePath() + "'",
Toast.LENGTH_SHORT).show();
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (SdpException e) {
    Toast.makeText(this, "SDP error:" +  e.getMessage(), Toast.LENGTH_SHORT).show();
    e.printStackTrace();
}

}
```

Load File from SDP

To load file from SDP, you can Use *SdpFileSystem.getFilesDir()* to get directory root in the encrypted filesystem. Just like when ypu want to write into Sdp, you must first unlock the SdpEngine.

```
// MainActivity.java
private String loadFile() {
      SdpFileSystem sdpFileSystem = null;
try {
    SdpEngine.getInstance().unlock(SDP_ALIAS, SDP_PASSWORD);
    sdpFileSystem = new SdpFileSystem(this, SDP_ALIAS);
} catch (SdpException e) {
    e.printStackTrace();
}
File file = new File(sdpFileSystem.getFilesDir(), "note.txt");
byte[] bytesArray = new byte[(int) file.length()];
try {
    FileInputStream fis = new FileInputStream(file);
    fis.read(bytesArray); //read file into bytes[]
    fis.close();

} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

try {
    return new String(bytesArray);
} catch (Exception e) {
    return "";
}
 }
```


100% 2:43 PM
Secure Memo
Save Note
Load Note
Use Fingerprint
Write something...


100% 3:42 PM
Secure Memo
my secure memo

Version History

- Version 1.0 – Initial Release, 8 August 2018
- Version 1.1 – SDP Updated, September 2018