provider loader
for Android

Whitepaper

Version 1.0

07/13/2016

Autor: certgate

Document History

Version	Date	Autor	Notes
1.0	13.07.2016	certgate	-

Content

1 Concept

The Android provider loader concept was written due to the following reason:

Contrary to the Oracle Java implementation Google's Android Java does not support the installation of static providers. A static provider is a provider that can be installed to the system and is then available to each application. This concept is quite nice because everyone can use this provider as soon as it is needed whithout having other dependencies. There is also a benefit regarding software update policies. When the provider is updated the programs using it don't need to be re-compiled. Unfortunately Android Java does not support this concept!

In order to support this concept as best as we can we designed the provider loader. It's task is to load the cgJCE providers into the context of an application without linking the cgJCE library / APK. The cgJCE is preinstalled once per device and can be updated at any time without re-compiling other programs. The programs using our cgJCE just have to link the quite small loader library which is less intended for changes and can then load and use our providers. In the following we have a deeper look at this concept.

1.1 How the loader works

Due to security reasons we designed our loader only being able to load a JCE package which was signed by us. So before the loader starts to load anything regarding the JCE it first checks wheter the installed cgJCE was signed with our certificate.

If that is the case the loader uses the applications class loader to load the cgJCE classes into its own context. After this step the loader calls a predefined function via reflection in order to load the cgJCE. When this process is finished our providers can be either retrieved by the system or by our loader. The "finished" signal can be received via an optional callback. Lets have a look at the two different loading procedures.

1.1.1 Manual usage of the loader

First we recommend to implement the LoaderCallback interface in order to receive the "onLoaded" event. This call is triggered as soon as the cgJCE finished loading. Loading itsself happens asynchrously i.e. the loader starts the loading process and returns immediately[1]. When the job has finished "onLoaded" is called[2].

We also recommend checking whether the cgJCE is installed by isProviderInstalled. Also check if the providers are already loaded by isProviderAvailable before calling the load function.

An example could look like this:

[1] in fact it is not completely but almost immediately since we have to preload a legacy provider in order to guarantee compatibilityness to programs using older versions of our JCE.
[2] when the callback is implemented and given as parameter on "load" call

```
public class LoaderSample implements LoaderCallback
{
Context _c;

public LoaderSample(Context c)
{
      this._c = c;
}

@Override
public void onLoaded()
{
      //loading the JCE has finished

      //get provider
      Provider p = Security.getProvider("CERTGATE");

      //do something with the provider
      …
}

public void startLoading() throws ProviderLoadingFailedException, Pro-
viderSignatureInvalidException, ProviderNotFoundException
{
      if (ProviderLoader.isProviderInstalled(this._c))
      {
            if (!ProviderLoader.isProviderAvailable())
            {
                  ProviderLoader.load(this._c, this);
            }
            else
            {
                  //already loaded!
            }
      }
      else
      {
            //not installed!
      }
}

}
```

1.1.2 Automated loading via JCEInitTask

For easier integration we implemented the JCEInitTask which extends from Android's AsyncTask and loads the cgJCE with more detailed success / failure information.

It is recommended to implement the OnTaskFinishCallback since it gives information about how the loading resulted.

Furthermore loading the cgJCE by this task gives the opportunity to fade-in a progress- or a custom dialog which dissapears after the loading has finished. This makes sense when the user shall be blocked or notified as long as the cgJCE is loading. The usage of this dialog is optional. When null is used as parameter a default progress dialog is shown to the user otherwise the custom dialog appears. There is also a constructor for not using any dialog.

Additionally there is the option to show message toasts when loading starts and finishes.

The following example shows how to use the JCEInitTask.

```
public class LoaderSample implements OnTaskFinishedCallback
{
Context _c;

public LoaderSample(Context c)
{
      this._c = c;
}

public void startLoading()
{
      new InitJCETask(this._c, null, this, false).execute();
}

@Override
public void onFinished(JCELoadResult result)
{
      switch (result)
      {
            case E_ALREADY_LOADED:
                  break;
            case E_NOT_INSTALLED:
                  break;
            case E_SUCCESS:
                  //get providers
                  ProviderLoader.getProviders();

                  break;
            case E_UNSPECIFIED_ERROR:
                  break;
      }
}

}
```

2 Features

Besides loading the cgJCE the loader has some other features e.g. receiving the registered providers. Furthermore the loader offers two wrapper classes which simulate some of the JCE SDK functions. Due to our loader concept and other classloader issues it is not possible to integrate our JCE SDK into the loader. In order to offer the most important functions regarding SmartCardAuthProviders we implemented the wrappers which use reflection calls to execute the desired functions. Lets have a look at the most important loader features.

2.1 ProviderLoader

- getProviders

  Returns a vector of SmartCardAuthProviderWrappers.

- load

  Loads the cgJCE and registers all available SmartCardAuthProviders.

- isProviderInstalled

  Returns whether the cgJCE APK is installed in the device.

- isProviderAvailable

  Returns whether the cgJCE as already loaded.

2.2 SmartCardAuthProviderWrapper

Each SmartCardAuthProvider registered by the cgJCE can be recevied by calling Security.getProvider. Calling that function results in returning the provider as Provider class[3]. Since the standard Java Provider class does not support our features we would like to cast the received instance to SmartCardAuthProvider. As already mentioned we are unable to include our JCE SDK into the loader due to classloader issues in conjunction with our loader concept. For this reason we implemented the SmartCardAuthProviderWrapper. This class is a wrapper for SmartCardAuthProviders and simulates their functionality. For SmartCardAuthProviderWrapper construction we need a SmartCardAuthProvider received by Security.getProvider which is returned as Provider. After constructing an instance of SmartCardAuthProviderWrapper all features of a SmartCardAuthProvider are available as wrapped reflection function calls. These functions are:

- login
- logout
- setCallbackHandler
- getName
- getServices
- registerSmartCardEventHandler

[3] when the provider exists / was registered

- unregisterSmartCardEventHandler
- getSmartCardExtension
- getProvider

All functions behave like specified in the JCE where getProvider is additionally. This function is used for receiving the original provider when the wrapper is not enough[4]. For more information regarding the functions see the JCE specification on www.oracle.com.

2.3 SmartCardExtensionWrapper

For the SmartCardExtensionWrapper applies the same as for the SmartCardAuthProviderWrapper – since we are unable to integrate the JCE SDK into the loader we need a wrapper for simulating the desired functions. The functions are:

- getRemainingAttemptsUser
- getRemainingAttemptsAdmin
- verifyPINUser
- verifyPINAdmin
- changePINUser
- changePINAdmin
- overwriteUserPIN
- getFirmwareVersion
- getManufacturerID
- getSerialNumber
- getCardType
- getReaderName
- getFreeMemorySizeByte
- getSupportsRNG
- isCardPresent

For more information regarding the functions see the cgJCE dokumentation.

[4] for instance for Cipher.getInstance(algoName, providerInstance) calls.