Android Explorations

Sidebar

Code signing in Android ...
Kanji Recognizer Premium Upgrade Hanzi Recognizer Premium Upgrade WWWJDIC for Android F

Android code signing
Secure USB deb... 7
Certificate pinnin... 8
Single sign-on to Googl...
Android online account ...
Emulating a PKI ... 49
Exploring Google... 28
Android secure el... 5
Accessing the em... 13
Changing Androi... 31
Certificate blacklisting in...
Jelly Bean hardw... 7
Using app encryption in ...
Hanzi Recognizer v2.2 ...
Unpacking Androi... 31
Storing applicatio... 19
Kanji Recognizer v2.2 R...
Using Password-... 39
WWWJDIC for Android 2.2
New Handwriting Recog...

Accessing the embedded secure element in Android 4.x

After discussing credential storage and Android's disk encry [http://nelenkov.blogspot.jp/2012/08/changing-androids-disk-encryption.html] we'll now look at another way to protect your secrets: the embe secure element (SE) found in recent devices. In the first post of this t part series we'll give some background info about the SE and show to use the SE communication interfaces Android 4.x offers. In the se part [http://nelenkov.blogspot.jp/2012/08/android-secure-element-execution we'll try sending some actual commands in order to find out more a the SE execution environment. Finally [http://nelenkov.blogspot.jp/20 /exploring-google-wallet-using-secure.html] we will discuss Google Wallet how it makes use of the SE.

What is a Secure Element and why do you want one?

A Secure Element (SE) is a tamper resistant smart [http://en.wikipedia.org/wiki/Smart_card] chip capable of running smart applications (called applets or cardlets) with a certain level of security features. A smart card is essentially a minimalistic comp environment on single chip, complete with a CPU, ROM, EEPROM, RAM and I/O port. Recent cards also come equipped with cryptographic co-processors implementing common algorithms such as DES, AES and RSA. Smart cards use various techniques to implement tamper resistance [http://en.wikipedia.org/wiki/Tamper_resistant] , making it quite hard to extract data by disassembling or analyzing the chip. They come pre-programmed with a multi-application OS that takes advantage of the hardware's memory protection features to ensure that each application's data is only available to itself. Application installation and (optionally) access is controlled by requiring the use of cryptographic keys for each operation.

The SE can be integrated in mobile devices in various form factors: UICC [http://en.wikipedia.org/wiki/UICC] (commonly known as a SIM card), embedded in the handset or connected to a SD card slot. If the device supports NFC [http://en.wikipedia.org/wiki/Near_field_communication] the SE is usually connected to the NFC chip, making it possible to communicate with the SE wirelessly.

Smart cards have been around for a while and are now used in applications ranging from pre-paid phone calls and transit ticketing to credit cards and VPN credential storage. Since an SE installed in a mobile device has equivalent or superior capabilities to that of a smart card, it can theoretically be used for any application physical smart cards are currently used for. Additionally, since an SE can host multiple

Send feedback

Android Explorations

Sidebar

Code signing in Android ...
Android code signing
Secure USB deb... 7
Certificate pinnin... 8
Single sign-on to Googl...
Android online account ...
Emulating a PKI ... 49
Exploring Google... 28
Android secure el... 5
Accessing the em... 13
Changing Androi... 31
Certificate blacklisting in...
Jelly Bean hardw... 7
Using app encryption in ...
Hanzi Recognizer v2.2 ...
Unpacking Androi... 31
Storing applicatio... 19
Kanji Recognizer v2.2 R...
Using Password-... 39
WWWJDIC for Android 2.2
New Handwriting Recog...

So a SE is obviously a very useful thing to have and with a lot of potential, but why would you want to access one from your apps? A from the obvious payment applications, which you couldn't realisti build unless you own a bank and have a contract with [http://corporate.visa.com/] and friends, there is the possibility of st other cards you already have (access cards, loyalty cards, etc.) on phone, but that too is somewhat of a gray area and may requ contracting the relevant issuing entities. The main application for party apps would be implementing and running a critical part of the such as credential storage or license verification inside the S guarantee that it is impervious to reversing and cracking. Other apps can benefit from being implemented in the SE are One Time Pass (OTP) generators and, of course PKI credential (i.e., private k storage. While implementing those apps is possible today with star tools and technologies, using them in practice on current comme Android devices is not that straightforward. We'll discuss this in deta second part of the series, but let's first explore the types of SEs avai on mobile devices, and the level of support they have in Android.

Secure Element form factors in mobile devices

As mentioned in the previous section, SEs come integrated in diffe flavours: as an UICC, embedded or as plug-in cards for an SD card slot. This post is obviously about the embedded SE, but let's briefly review the rest as well.

Pretty much any mobile device nowadays has an UICC (aka SIM card, although it is technically a SIM only when used on GSM networks) of some form or another. UICCs are actually smart cards that can host applications, and as such are one form of a SE. However, since the UICC is only connected to the basedband processor, which is separate from the application processor that runs the main device OS, they cannot be accessed directly from Android. All communication needs to go through the Radio Interface Layer (RIL) which is essentially a proprietary IPC interface to the baseband. Communication to the UICC SE is carried out using special extended AT commands (`AT+CCHO`, `AT+CCHC`, `AT+CGLA` as defined by 3GPP TS 27.007 [http://www.3gpp.org/ftp/Specs/html-info/27007.htm]), which the current Android telephony manager does not support. The SEEK for Android [http://code.google.com/p/seek-for-android/] project provides patches that do implement the needed commands, allowing for communicating with the UICC via their standard SmartCard API [http://seek-for-android.googlecode.com/svn/trunk/doc/index.html] , which is a reference implementation of the SIMalliance [http://www.simalliance.org/] Open Mobile API [http://www.simalliance.org/en/about/workgroups/open_mobile_api_working_group/] specification. However, as most components that talk directly to the hardware in Android, the RIL consists of an open source part (`rild`), and a proprietary library (`libXXX-`

Android Explorations

experiments. While there is some talk of integrating this functionality into stock Android (there is even an empty packages/a /SmartCardService directory in the AOSP tree), there is current standard way to communicate with the UICC SE through the RIL (s commercial devices with custom firmware are rep [http://code.google.com/p/seek-for-android/wiki/DeviceDetails] to suppo though).

An alternative way to use the UICC as a SE is using the Single Protocol (SWP [http://en.wikipedia.org/wiki/Single_Wire_Protocol]) wher UICC is connected to a NFC controller that supports it. This is the ca the Nexus S, as well as the Galaxy Nexus, and while this functional supported by the NFC controller drivers, it is disabled by default. Tl however a software limitation, and people have managed to p [http://forum.xda-developers.com/showthread.php?t=1281946] AOSP sour get around it and successfully communicate with UICC. This has greatest potential to become part of stock Android, however, as o current release (4.1.1), it is still not available.

Another form factor for an SE is an Advanced Security SD card (A [https://www.sdcard.org/developers/overview/ASSD/]), which is basicall SD card with an embedded SE chip. When connected to an An device with and SD card slot, running a SEEK-patched Android version, the SE can be accessed via the SmartCard API. However, Android devices with an SD card slot are becoming the exceptions rather than the norm, so it is unlikely that ASSD Android support will make it to the mainstream.

And finally, there is the embedded SE. As the name implies, an embedded SE is part of the device's mainboard, either as a dedicated chip or integrated with the NFC one, and is not removable. The first Android device to feature an embedded SE was the Nexus S, which also introduced NFC support to Android. Subsequent Nexus-branded devices, as well as other popular handsets have continued this trend. The device we'll use in our experiments, the Galaxy Nexus, is built [http://www.ifixit.com/Teardown/Samsung-Galaxy-Nexus-Teardown/7182/2] with NXP's PN65N [http://www.nxp.com/news/press-releases/2011/11/nxp-nfc-solution-implemented-in-galaxy-nexus-from-google.html] chip, which bundles a NFC radio controller and an SE (P5CN072 [http://www.classic.nxp.com/acrobat_download2/other/identification/SFS107710.pdf] , part of NXP's SmartMX [http://mifare.net/files/3013/0079/2103/SmartMX%20Leaflet_Oct10.pdf] series) in a single package (a diagram can be found here [http://www.nfc.cc/technology/nxp-nfc-chips/]).

NFC and the Secure Element

NFC and the SE are tightly integrated in Android, and not only because

Android Explorations

Kanji Recognizer Premium Upgrade Hanzi Recognizer Premium Upgrade WWWJDIC for Android F

- card emulation (CE) mode, which allows the device to emulate a traditional contactless smart card

What can Android do in each of these modes? The R/W mode allows to read NDEF tags and contactless cards, such as some transport c While this is, of course, useful, it essential turns your phone ir glorified card reader. P2P mode has been the most demoed marketed one, in the form of Android Beam [http://developer.androic /guide/topics/connectivity/nfc/nfc.html#p2p] . This is only cool the first cc of times though, and since the API only gives you higher-level acce the underlying P2P communication protocol, its applications are curr limited. CE was not available in the initial Gingerbread release, and introduced later in order to support Google Wallet [http://www.google /wallet/] . This is the NFC mode with the greatest potential for rea applications. It allows your phone to be programmed to emulate p much any physical contactless card, considerably slimming down physical wallet in the process.

The embedded SE is connected to the NFC controller throug SignalIn/SignalOut Connection (S2C, standardized as NF([http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-373.pc and has three modes of operation: off, wired and virtual mode. I mode there is no communication with the SE. In wired mode the SE is visible to the Android OS as if it were a contactless smartcard connected to the RF reader. In virtual mode the SE is visible to external readers as if the phone were a contactless smartcard. These modes are naturally mutually exclusive, so we can communicate with the SE either via the contactless interface (e.g., from an external reader), or through the wired interface (e.g., from an Android app). This post will focus on using the wired mode to communicate with the SE from an app. Communicating via NFC is no different than reading a physical contactless card and we'll touch on it briefly in the last post of the series.

Accessing the embedded Secure Element

This is a lot of (useful?) information, but we still haven't answered the main question of this entry: how can we access the embedded SE? The bad news is that there is no public Android SDK API for this (yet). The good news is that accessing it in a standard and (somewhat) officially supported way is possible in current Android versions.

Card emulation, and consequently, internal APIs for accessing the embedded SE were introduced in Android 2.3.4, and that is the version Google Wallet launched on. Those APIs were, and remain, hidden from SDK applications. Additionally using them required system-level permissions (`WRITE_SECURE_SETTINGS` or `NFCEE_ADMIN`) in 2.3.4 and subsequent Gingerbread releases, as well as in the initial Ice Cream Sandwich release (4.0, API Level 14). What this means is that only

Send feedback

Android Explorations

Sidebar

carrier), this was good enough. However, it made it impossible to develop and distribute an SE app without having it signed by the platform ve Android 4.0.4 (API Level 15) changed that by replacing the system-permission requirement with signing certificate (aka, 'signature' in An framework terms) whitelisting at the OS level. While this still req modifying core OS files, and thus vendor cooperation, there is no ne sign SE applications with the vendor key, which greatly simp distribution. Additionally, since the whiltelist is maintained in a file, i easily be updated using an OTA to add support for more SE applicati

In practice this is implemented by the `NfceeAccessControl` class enforced by the system `NfcService`. `NfceeAccessControl` read whilelist from `/etc/nfcee_access.xml` which is an XML file stores a list of signing certificates and package names that are allow access the SE. Access can be granted both to all apps signed particular certificate's private key (if no package is specified), or single package (app) only. Here's how the file looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:xliff="urn:oasis:names:tc:xliff:dc
  <signer android:signature="30820...90">
    <package android:name="org.foo.nfc.app">
  </package></signer>
</resources>
```

This would allow SE access to the 'org.foo.nfc.app' package, if it is signed by the specified signer. So the first step to getting our app to access the SE is adding its signing certificate and package name to the `nfcee_access.xml` file. This file resides on the system partition (`/etc` is symlinked to `/system/etc`), so we need root access in order to remount it read-write and modify the file. The stock file already has the Google Wallet certificate in it, so it is a good idea to start with that and add our own package, otherwise Google Wallet SE access would be disabled. The 'signature' attribute is a hex encoding of the signing certificate in DER format, which is a pity since that results in an excessively long string (a hash of the certificate would have sufficed) . We can either add a <debug/> element to the file, install it, try to access the SE and get the string we need to add from the access denied exception, or simplify the process a bit by preparing the string in advance. We can get the certificate bytes in hex format with a command like this:

```
$ keytool -exportcert -v -keystore my.keystore -alias my_
-storepass password|xxd -p -|tr -d '\n'
```

This will print the hex string on a single line, so you might want to redirect

Android Explorations

As we said, there are no special permissions required to access the S ICS (4.0.3 and above) and Jelly Bean (4.1), so we only need to add standard NFC permission to our app's manifest. However, the library implements SE access is marked as optional, and to get it loaded fo app, we need to mark it as required in the manifest with the <us library> tag. The AndroidManifest.xml for the app sh look something like this:

```
<manifest xmlns:android="http://schemas.android.com
    package="org.foo.nfc.app"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="16" />

    <uses-permission android:name="android.permissi

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.acti
                <category android:name="android.intent.ca
            </intent-filter>
        </activity>

        <uses-library
            android:name="com.android.nfc_extras"
            android:required="true" />
    </application>
</manifest>
```

With the boilerplate out of the way it is finally time to actually access the SE API. Android doesn't currently implement a standard smart card communication API such as JSR 177 [http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/security/satsa-api/jsr177/index.html] or the Open Mobile API [http://www.simalliance.org/en/about/workgroups/open_mobile_api_working_group/] , but instead offers a very basic communication interface in the NfcExecutionEnvironment (NFC-EE) class. It has only three public methods:

Android Explorations

Kanji Recognizer Premium Upgrade   Hanzi Recognizer Premium Upgrade   WWWJDIC for Android F...

```
    public void close() throws IOException {...}

    public byte[] transceive(byte[] in) throws IOEx
}
```

This simple interface is sufficient to communicate with the SE, so no just need to get access to an instance. This is available via a s method of the `NfcAdapterExtras` class which controls both emulation route (currently only to the SE, since UICC support is available) and NFC-EE management. So the full code to ser command to the SE becomes:

```
NfcAdapterExtras adapterExtras = NfcAdapterExtras.g
NfcExecutionEnvironment nfceEe = adapterExtras.getE
nfcEe.open();
byte[] response = nfcEe.transceive(command);
nfcEe.close();
```

As we mentioned earlier however, `com.android.nfc_extras` i optional package and thus not part of the SDK. We can't import it dir so we have to either build our app as part of the full Android source placing it in `/packages/apps/`), or resort to reflection. Since the SE interface is quite small, we opt for ease of building and testing, and will use reflection. The code to get, open and use an NFC-EE instance now degenerates to something like this:

```
Class nfcExtrasClazz = Class.forName("com.android.nfc_ext
Method getMethod = nfcExtrasClazz .getMethod("get", Class
NfcAdapter adapter = NfcAdapter.getDefaultAdapter(context
Object nfcExtras = getMethod .invoke(nfcExtrasClazz, adap

Method getEEMethod = nfcExtras.getClass().getMethod("getE
                    (Class[]) null);
Object ee = getEEMethod.invoke(nfcExtras , (Object[]) nul
Class eeClazz = se.getClass();
Method openMethod = eeClazz.getMethod("open", (Class[]) n
Method transceiveMethod = ee.getClass().getMethod("transc
                    new Class[] { byte[].class });
Method closeMethod = eeClazz.getMethod("close", (Class[])

openMethod.invoke(se, (Object[]) null);
Object response = transceiveMethod.invoke(se, command);
closeMethod.invoke(se, (Object[]) null);
```

We can of course wrap this up in a prettier package, and we will in the

Send feedback

Android Explorations


```
D/SEConnection(27318): --> 00000000
D/SEConnection(27318): <-- 6E00
```

We'll explain what the response means and show how to send s actually meaningful commands in the second part of the article.

Summary

A secure element is a tamper resistant execution environment on a that can execute applications and store data in a secure manner. A is found on the UICC of every Android phone, but the platform curr doesn't allow access to it. Recent devices come with NFC support, v is often combined with an embedded secure element chip, usually i same package. The embedded secure element can be accessed externally via a NFC reader/writer (virtual mode) or internally vi `NfcExecutionEnvironment` API (wired mode). Access to the A currently controlled by a system level whitelist of signing certificates package names. Once an application is whitelisted, it can commun with the SE without any other special permissions or restrictions.

Posted 22nd August 2012 by Nikolay Elenkov

Labels: android security

View comments

Send feedback