Distribution: EW

Revision of this document:			
Revision	Data:	**Reason for edition, change:**	Author:
01	06.12.2019	Initial release	H. Bihr
02			
03			
04			

Freigabe von xx: **Datum:** **Unterschrift:**	**Freigabe von xx:** **Datum:** **Unterschrift:**	**Freigabe von QM:** **Datum:** **Unterschrift**

Table of contents

Revision of this document: 1
1 Purpose 2
2 Apps on Android based payment terminals 2
3 Card reader interface 2
3.1 Technologies used 2
3.1.1 Bound service and Android Messenger 2
3.1.2 Google protocol buffers 2
3.2 Interface description 2
3.2.1 Bound service 2
3.2.2 Message from third party apps to payment application 3
3.2.3 Messages returned by the CCV payment application 3
3.2.4 Protobuf definition 3
3.3 Access restrictions 3
3.3.1 Mutual exclusion 3
3.3.2 Whitelists 4
4 Examples 7
5 References 7

1 Purpose

This document describes an interface available to app developers on Android-based payment terminals offered by CCV. This interface shall provide access to both customer card reader and SAM slots of a payment terminal. App developers shall use this interface for transactions initiated directly by their apps.

2 Apps on Android based payment terminals

Android-based payment terminals can run many apps in parallel. CCV will always install a payment application on these terminals. Third party developers may develop additional apps for these devices. These apps are available to customers for installation.

For several reasons, including security, only the payment application provided by CCV has direct access to the card reader of the hardware device. Third party apps will have to make requests to the payment application if they want to access the card reader

This API document defines a client (third party apps) and server (payment application) relationship. All requests are synchronous calls; this means for every request to the payment application there will always be a corresponding response message sent back to the client.
The payment application as central server ensures that are no conflicting requests from multiple client apps.

3 Card reader interface

3.1 Technologies used

3.1.1 Bound service and Android Messenger

For communication between third party apps and the payment application, the latter provides a bound service [1]. A Messenger is used to transfer actual message data [2].
In order to forward response messages to the correct recipient, every client app shall provide a Messenger of its own in every request message.

3.1.2 Google protocol buffers

Google protocol buffers [3] allow serializing complex data structures into byte streams that can be passed from one application to another. Google protocol buffers contain a compiler which, based on a .proto definition file, creates libraries for apps that provide an API for manipulating and evaluating data structures defined in the .proto file. The compiler can generate these libraries and corresponding headers for many programming languages.
All messages transported over this interface contain serialized protocol buffer byte streams as content.
CCV makes the .proto definition file available to app developers.

3.2 Interface description

3.2.1 Bound service

The service offered by the payment application is accessible by its name
`eu.ccv.de.android.cardreader.BIND`.
Every message contains a Bundle of data.

3.2.2 Message from third party apps to payment application

These messages have to include the following fields in the data Bundle:

Key	Data type	Description
eu.ccv.de.android.cardreader.extra.RESULT_RECEIVER	IBinder	Provides the IBinder of the app that is the recipient of the response
eu.ccv.de.android.cardreader.extra.REQUEST	byte[]	Contains the serialized protocol buffer message of the request
eu.ccv.de.android.cardreader.extra.FINAL_TRX_MSG_FLAG	boolean	Value “true” identifies this as the last request in a sequence. After processing this call, the interface is again available for other apps

3.2.3 Messages returned by the CCV payment application

These messages contain the following field:

Key	Data type	Description
eu.ccv.de.android.cardreader.extra.RESPONSE	byte[]	Contains the serialized protocol buffer response message

3.2.4 Protobuf definition

The .proto definition file is available at [4].
This file defines two types of messages:

- `ReqPinPad` is sent from third party apps to the payment application
- `ResPinPad` is sent from the payment application to third party apps as response

A ResPinPad message contains either an error (`EPinPadResultCode`) or a response matching the request.
For example, the response to a `ReqCrdReadData` will contain a `ResCrdReadData`.

3.3 Access restrictions

3.3.1 Mutual exclusion

As the payment application can handle only one request at a time, it will implement a mutual exclusion mechanism that allows only a single app to access the card reader at any time.
Apps may issue several commands that logically belong together; therefore, a session management ensures that an app has exclusive access to the card reader for a whole sequence of commands.

Sessions handling works according to the following rules:

- A client opens a new session implicitly with the first request that it sends.

- The payment application identifies a client and its session by the value of in `eu.ccv.de.android.cardreader.extra.RESULT_RECEIVER` in the request message. Therefore, within a session, the value of this field must not change.
- A client can close a session by sending a message with `eu.ccv.de.android.cardreader.extra.FINAL_TRX_MSG_FLAG` set to true.
- A session automatically times out after 30 seconds of inactivity. In case of such a timeout, the payment application will power off the NFC antenna and contacted chip cards.

While one app has opened a session, the payment application will reject requests from all other apps with a response in which EPinPadResult has the value DEVICE_BUSY.

3.3.2 Whitelists

To prevent third party apps from automatically becoming part of the PCI DSS scope, the payment application contains two whitelist files:

- AID whitelist (for chip applications)
- BIN range whitelist (for magnetic stripe)

The payment application will mask magnetic card data from any card that is not in the BIN range whitelist.
The payment application will block “select AID” chip commands if the particular AID is not in the AID whitelist.

With this mechanism, it is possible to keep this interface free of card data that is in PCI scope.

CCV controls the content of these whitelist files. Consequently, app developers will have to report the AIDs and BIN ranges they use to CCV for adding these cards to the whitelists. CCV needs a card specification for this to ensure that there is not overlap with cards in PCI scope. It is the responsibility of the app developer to provide this specification.

4 Command Description

In the following section the commands that can be send using the protobuf protocol structure are described. To see a full description of all commands and parametes have a look at the proto file.

4.1 Read Card Data (ReqCrdReadData)

This method is used to read data from a card. It reads the ATR of an ICC/CTLS card and/or the track data of a magnetic stripe based card. The method can be used to wait for card entry.

Remark: Note that after the card read the readers are disabled. If you want to use the card for APDU exchange you first have to power on the reader again.

ReqCrdReadData
Int32 crd_types: Slots (Card types (bit pattern)) where a card should be waited for
int32 timeout: Timeout. Defines the timeout to wait for card entry.

ResCrdReadData
ECardTypes crd_type: Card Type that was read
CrdData crd_data: Card data that was read

4.2 Is Card Inserted (ReqCrdIsInserted)

This method provides the information, if a card is inserted in the ICC reader.
The method is non-applicable for SAM slots

ReqCrdIsInserted
no parameters defined

ResCrdIsInserted
ECardTypes crd_type: Slots (identified by card type) where a cound was found

4.3 Power Icc/Ctls On (ReqCrdIccPowerOn)

This method applies power to the selected ICC card reader slot
and performs a cold reset.

ReqCrdIccPowerOn
ECardTypes crd_type: Slot (identified by card type) that should be powered.

ResCrdIccPowerOn
string atr: ICC Answer to reset. Available, if an ICC card is inserted.

4.4 Warm reset card

This method resets the selected ICC card and performs a warm reset

ReqCrdIccWarmReset
ECardTypes crd_type: Slot (identified by card type) where a warm reset should be performed.

ResCrdIccWarmReset
string atr: ICC Answer to reset. Available, if an ICC card is inserted.

4.5 Power Icc/Ctls Off (ReqCrdIccPowerOff)

This method removes power from the selected ICC card reader slot

ReqCrdIccPowerOff
ECardTypes crd_type: Slot (identified by card type) that should be powered off

4.6 Send APDU (ReqCrdIccSendApdu)

This method sends an APDU to the ICC card reader slot appropriate for the selected card type and returns the answer from the card

ReqCrdIccSendApdu

ECardTypes crd_type: Card type to which APDU should be send
string icc_apdu_req: APDU request for card

ResCrdIccSendApdu
string icc_apdu_res: APDU request from card

4.7 Mifare Read Block (ReqCrdMifareReadBlock)

Request to read Mifare card data. This only applies to basis Mifare Classic commands. For more complex Mifare commands please refer to Send APDU. Mifare Classic 1K uses 1 kB, organized in 16 sectors of 4 blocks

ReqCrdMifareReadBlock

int32 mifare_block_num: Identifies a block (segment) in the memory. One block consists of 16 byte).
int32 mifare_num_of_blocks: Identifies the number of blocks (segments) that should be read. If -1 is passed, read till last block.
int32 mifare_key_type: Identifies the key type for the authentication procedure.
int32 mifare_key_index: Identifies the key index for the authentication procedure.

ResCrdMifareReadBlock
string mifare_data_block: Contains data blocks that are read from the Mifare card.

4.8 Mifare Read Block (ReqCrdMifareWriteBlock)

Request to write Mifare card data. This only applies to basis Mifare commands. For more complex Mifare commands please refer to Send APDU

ReqCrdMifareWriteBlock

int32 mifare_block_num: Mifare block number. Identifies a block (segment) in the memory of a Mifare card.
int32 mifare_key_type: Identifies the key type for the authentication procedure.
int32 mifare_key_index: Identifies the key index for the authentication procedure.
string mifare_data_block: Contains data blocks that should be written to the Mifare card. Data length can be larger than single Mifare block to write multiple consecutive blocks.

ResCrdMifareWriteBlock
No parameters defineds

4.9 Mifare (ReqCrdMifareLoadKey)

This method loads an authentication key for Mifare cards.
The keys are loaded in clear text or encrypted according to the parameter encryption type. This only applies to basis Mifare commands. For more complex Mifare commands please refer to Send APDU

ReqCrdMifareLoadKey
int32 mifare_key_index: Identifies the key index for the authentication procedure.
string mifare_key_data: Contains a clear text or encrypted key. The encryption mode is defined by the mifare_key_crypt_mode tag.
int32 mifare_key_crypt_mode: Defines the encryption mode of a Mifare key (tag mifare_key_data).

ResCrdMifareLoadKey

No parametes defined

5 Examples

Binding the Service:

```
   private final ServiceConnection serviceConnection =  new ServiceConnection()

   private void bindReaderService(){
      List<String> packageNames = getRegisteredPackageNames("eu.ccv.de.android.cardreader.BIND");
      Intent bindServiceIntent = new Intent("eu.ccv.de.android.cardreader.BIND");

      if (packageNames.size() > 0) {
         bindServiceIntent.setPackage(packageNames.get(0));
         getApplicationContext().bindService(bindServiceIntent, serviceConnection,
Context.BIND_AUTO_CREATE);
      }
   }
```

Sample for Cardreader Request (Power On)

```
AndroidCardreader.ReqPinPad request =
AndroidCardreader.ReqPinPad.newBuilder().setPowerOn(AndroidCardreader.ReqCrdIccPowerOn.newBuilder()
.setCrdType(cardType).build()).build();
 Message message = Message.obtain();
 Bundle bundle = new Bundle();
 bundle.putBinder(EXTRA_RETURN_CHANNEL, ownMessenger.getBinder());
 bundle.putByteArray(EXTRA_REQUEST, request.toByteArray());
 bundle.putBoolean(EXTRA_FINAL_TRX_MSG_FLAG, bFinal);
```

6 References

[1] https://developer.android.com/guide/components/bound-services.html
[2] https://developer.android.com/guide/components/bound-services.html#Messenger
[3] https://developers.google.com/protocol-buffers