भारतीय राष्ट्रीय भुगतान निगम

NATIONAL PAYMENTS CORPORATION OF INDIA

UNIFIED PAYMENTS INTERFACE

COMMON LIBRARY SPECIFICATIONS

Version 1.7

UNIFIED PAYMENTS INTERFACE

Document History

Change No.	Changes done	Revision / Version Number	Date of Change	Change Description
1	Initial Document	1.2	22-03-2016	
2	Addition of UPI CL 1.5 features	1.5	01-12-2016	OTP Auto Read, Resend OTP & ATM pin acceptance, Multilingual & Virtual Keypad details has been included in the page no 16.
3	List of FAQs added	1.5	01-12-2016	Bank frequent queries has been included in the page no 24.
4	Addition of UPI CL 1.7 new features	1.7	06-05-2021	1. Capturing of card details in the CL page 2. Capturing of extra device details inside CL 3. Forget UPI Pin feature 4. Resend OTP feature 5. Support for multiple languages in CL 6. Aadhaar OTP Authentication

Table of Contents

1. Introduction 5
2. Objectives 5
3. Components 5
   3.1 Common Library files 5
   3.2 Steps to integrate CL sdk file to psp app 5
4. Specification 6
   4.1 Prerequisites 6
5. Registration Process Steps 7
   5.1 Get Challenge Method 7
      5.1.1 Request 7
      5.1.2 Response 8
   5.2 Get Token Method 8
      5.2.1 Request 8
      5.2.2 Response 8
   5.3 Register App Method 9
      5.3.1 Request 9
      5.3.2 Response 10
   5.4 Get Credential Method 10
      5.4.1 Request 10
      5.4.2 Response 15
6 Rotation of Token 19
7 Get Credential service (User interface) 20
8 New feature of UPI CL 1.5 24
   8.1 OTP Auto detection 24
   8.2 ATMPIN + OTP Flow in Registration 25
   8.3 Virtual/Custom Numeric Keyboard 26
9 New feature of UPI CL 1.7 26
   9.1 Multilingual Support 26
   9.2 Verified Merchant Confirmation 26
   9.3 Resend OTP 27
   9.5 Capturing of debit card details and expiry date on CL page 29
      9.5.1 Set/Change UPI Pin 30

9.5.2 Aadhaar OTP Authentication 33
9.6 Capturing of device details. 34
10. Encryption Algorithm 36
11. Common Library Process Flow 36
12. Generation of XML payload in PSP server 37
13. Versioning Strategy 37
14. Configuration Parameters in Common Library 37
15. Common library release and distribution 38
16. In-app payment/Deep-linking from merchant apps 38
18.1 Recommendation on parameters 39
17. Deep-linking configuration guidelines for Android 39
1. Android Manifest 39
2. Pay Activity 39
3. PSP URL 40
4. On click listener of "Pay by UPI (using same mobile)" button 40
18. List of FAQs 41
19. Contact Us 42
20. Annexure-1 42
21.1MainActivity.java 42
21.2 CryptLib.java 49

1. Introduction

NPCI UPI common library (CL) is a specification with default implementation which PSP can embed into their UPI app to communicate with NPCI central system to set/reset/change UPI PIN (earlier known as MPIN), balance enquiry and for debit authorization with two factor secured authentication.

Common library ensures all sensitive data encrypted with latest technology encryption standards.

These libraries will be available for two major mobile operating systems such as Android and iOS. This document explains the configuration and implementation of common library services with UPI app for Android devices.

2. Objectives

- Provide a common interface to interact with NPCI UPI system to protect the user credentials
- Protect user credentials from replay and transfer credential.
- Bind PSP app with NPCI CL and vice-versa
- CL Captures the Credentials and protects for transport
- NPCI UPI server validate and authenticates the user credential.
- Provide higher end security component.

3. Components

3.1 Common Library files

1. Android Security Component (Securecomponent-release-signed_{version}.aar)

3.2 Steps to integrate CL sdk file to psp app

Verifying signed module

This step is to verify the common library authenticity that it is signed by NPCI
Note: Please make sure that Java Development Kit (JDK), Version 8 is installed in the system and the Path environment variable is having <JDK Installation>/bin folder entry. Jarsigner utility comes bundled with JDK.

Command:

```
jarsigner -verify   <FileName>
```

Sample:

```
jarsigner  -verify Securecomponent-release-signed..aar
```

Expected output:

Jar file Verified.

Warnings can be ignored. After verification add the module by following steps:

1. Go to File>New>New Module
2. Select "Import .JAR/. AAR Package" and click next.
3. Enter the path to .aar/.jar file and click finish.
4. Go to File>Project Structure (Ctrl+Shift+Alt+S).
5. Under "Modules," in left menu, select "app."
6. Go to "Dependencies tab.
7. Click the green "+" in the upper right corner.
8. Select "Module Dependency"
9. Select the new added module (SecureComponent-release-signed.aar) from the list.

4. Specification

Common library provides an interface to mobile applications for both financial & non-financial transactions that can be performed between PSP/Banks. Integration methodology and flow would differ depending upon the programming platform. Interface details are elaborated below.

Common Library provides three main interfaces such as:

- Registration
- Get Credential
- Rotation

The above three interfaces ensure the binding between NPCI UPI CL and PSP application

It is critical and important to fetch token from UPI after successful registration of the user with respective PSP.

It is recommended that the PSP rotates the token once every 90days

4.1 Prerequisites

PSP mobile app would use the following interface to communicate with common library.

Process Name	Details
Registration	During the initialization, common library will register with PSP mobile application and device. After successful registration, PSP mobile App will have a valid token. Without a valid token, Common library will not process any transaction requests.
Get Credential	This process captures sensitive information, encrypts and forwards encrypted data to PSP Application.
Rotation	This process will rotate the token once a month, If not the token would expire. Procedure and services will be same as Registration process, except for a few parameters.

5. Registration Process Steps

This steps will be processed at the time of registration which means that the NPCI CL is invoked by the PSP at the first time. Steps to be followed by PSP app to register with the common library,

Service Name	Details
Get Challenge	Execute "getChallenge" service to receive a challenge from common library.
Get Token	Use generated challenge in PSP server request to receive a valid token from UPI system using "*Get Token*" service (Same as ListKeys API with specific values in creds block). Received token should be stored in PSP mobile app to ensure secure communication with common library. Once GetToken is completed, RegisterApp to be executed immediately.
Register App	Execute "*Register App*" service to verify and register the generated token in the common library.

5.1 Get Challenge Method

PSP mobile app should execute "*Get Challenge*" service to receive a challenge from common library. PSP does a RPC to CL and asks for "Challenge" which can be decrypted only by NPCI server.

Initiated Class : clServices

Initiated Method : getChallenge

Method Name	Inputs	Data Type
getChallenge	device_id & type	Both are strings

5.1.1 Request

The PSP application should send "device_id" and "type" as input parameters to CL. Parameter to be passed are elaborated below:

Parameter Name	Data Type	Description
Device Id	String	This is a mandatory parameter specifying the current device id. (Android id will replace deviceid as part of Android X OS, since device id will not be available as part of Android X)
Type	String	This is a mandatory parameter specifying the requirement for getting challenge value. Values to be populated- ***"initial"*** for registration and ***"rotate"*** for rotation.

5.1.2 Response

Output of the above service will be a string containing the *"Challenge"*. PSP Mobile app should pass the generated challenge to PSP server to receive a valid token from UPI. If Common Library fails to generate a challenge, the output will return null.CL will also add version along with challenge starting from CL 1.5.2 version.

Sample output: "2.2|Challenge"

5.2 Get Token Method

PSP mobile app sends ReqListKeys api via psp which contains type="GetToken" and also passed the CL generated challenge to get the valid token value from UPI. UPI validates the challenges and return the key value as a response.

Initiated Service: ReqListKeys

5.2.1 Request

The mobile app should send the request via psp as "GetToken (device-id+ app-id+ mobile-number+ challenge)" to UPI.

```
<upi:ReqListKeys xmlns:upi="http://npci.org/upi/schema/">
      <Head ver="1.0" ts="" orgId="" msgId=""/>
      <Txn id="" note="" refId="" refUrl="" ts="" type="GetToken"/>
          <Creds>
                  <Cred type="Challenge" subtype="initial/reset">
                  <data code=""
                  ki="">device id|app id|mobile number|challenge</data>
                  //Note: device id, app id, mobile number and challenge are
                  concatenated including pipe characters in between.
                  </Cred>
          </Creds>
          </upi:ReqListKeys>
```

5.2.2 Response

The RespListKeys returns the following:

```
<key code="NPCI" type="PKI" ki="yyyymmdd">
   <keyValue>base64 encoded token</keyValue>
</key>
```

Here the highlighted part is the token in a Base64 encoded format. This can be decoded to Byte Array and encoded again as Hex String. For all subsequent communication to NPCI Library, this Hex String is to be used as a token.

5.3 Register App Method

After receiving a valid token from UPI, PSP mobile app should store it in the application. After which "Register App (hmac, device -id, app-id, mobile-number)" RPC service should be executed to validate the token and register the PSP app.

5.3.1 Request

Initiated Method: clServices

Method Name	Inputs	Data Type
registerApp	hmac	String
	device-id	String (Android id will replace deviceid as part of Android X OS, since device id will not be available as part of Android X)
	app-id	String
	Mobile-number	String
	random	String

Parameters to be passed are elaborated below.

Parameter Name	Data Type	Description
Device-Id	String	This is a mandatory parameter for specifying the current device id. (Android id will replace deviceid as part of Android X OS, since device id will not be available as part of Android X)
App-Id	String	This is a mandatory parameter for specifying the current app id.
Mobile-number	String	This is a mandatory parameter for specifying the mobile number of the user which needs to be verified by the PSP app.

Hmac	String	This is a mandatory parameter. Steps to be followed to generate HMAC. a. Concatenate device-id, app-id and mobile-number with the separator "|". b. Create a 16-byte random values. c. Create a hash of the concatenated string using SHA-256 algorithm and random created earlier. d. Encrypt the hash with the token as key using AES-256 algorithm. Use random as iv parameter. e. Populate Hmac with the encrypted string
random	String	This is the Base64 encoded 16byte random string generated by app. Which is used as random for creating hash and also as iv parameter while AES encryption of hash for Hmac.

5.3.2 Response

A Boolean value will be generated using the above request representing success (1) or failure (0) of registration.

UNIFIED PAYMENTS INTERFACE

5.4 Get Credential Method

PSP mobile app should execute the "*Get Credential*" service of common Library to capture sensitive information such as MPIN , ATMPIN or OTP.

5.4.1 Request

The psp initiates "Get Credential" service of CL to get all the trusted values

Initialized Method: clServices

Method Name	Inputs	Data Type
getCredential	keyCode	String
	xmlPayload	String
	controls	String
	Configuration	String
	Salt	String
	Trust	String
	payInfo	String
	languagePref	String

Parameters are elaborated below

Parameter Name	Data Type	Description
keyCode	String	This is a mandatory parameter specifying NPCI or UIDAI public key is being used. Valid values can be NPCI or UIDAI.
xmlPayload	String	This is a mandatory field containing the digitally signed XML payload received from list-Keys API of UPI. Response of list-keys API not to be modified.

controls	String	This field specifies the schema for the credential(s) to be captured, which is in JSON String Format. Common library reads this data and generates the Controls. The list of these credentials are to be captured by the PSP app from ListAccountResponse API. Example is as below:
		{
		"CredAllowed": [
		{
		"type": "OTP",
		"subtype": "SMS",
		"dType": "NUM",
		"dLength": "6"
		},
		{
		"type": "PIN",
		"subtype": "ATMPIN",
		"dType": "NUM",
		"dLength": "4"
		}, //will be part if FORMAT2
		{
		"type": "PIN",
		"subtype": "MPIN",
		"dType": "NUM",
		"dLength": "6"
		},
		{
		"type": "OTP",
		"subtype": "AADHAAR",
		"dType": "NUM",
		"dLength": "6"
		}//Only if Aadhar flow
		]
		}
		
		1.If the bank supports FORMAT1 ATMPIN Data block will not be there.

		2.If Bank supports FORMAT2, ATMPIN data block should be part of the credAllowed block. 3.If the bank supports" FORMAT3|FORMAT2" or "FORMAT3|FORMAT1" functionality and Aadhaar consent provided in the app, Aadhar OTP data block should be part of the credAllowed block [CL will capture Aadhar OTP, Issuer OTP and MPIN]. And if Aadhar consent is not provided app has to use fallback mechanism and use either FORMAT1 or FORMAT2(based on fallback format received), in this case card details will be captured inside the CL. Based on the number of blocks in the JSON, one or more credential input control will be rendered by the common library.
Configuration	String	This is an optional parameter to customize the UI displayed by the common library. New features of Verified Merchant, Resend OTP and Forgot UPI PIN are available as part of CL 1.7. Sample JSON for the same is provided below: {"payerBankName":"Indian Bank Ltd.", "imageUrl": "https://www.someurl.com/bankimage.jpg", "resendAadhaarOTPFeature": "true", "resendIssuerOTPFeature": "false", "issuerResendOTPLimit": "2", "aadhaarResendOTPLimit": "1", "verifiedMerchant": "true", "forgotUpiPINEnabled": "true","captureCardDetails":"true"} Refer to Section 9.2 for Verified Merchant Confirmation details Refer to Section 9.3 for Resend OTP Refer to Section 9.4 for Forgot UPI PIN details captureCardDetails: for FORMAT2 and FORMAT1 this tag is not applicable and should not be sent. This value needs to be provided as true if the bank Supports "FORMAT3|FORMAT2" or "FORMAT3|FORMAT1" and Aadhaar consent not provided in the app (capturing card details inside CL page), Else provide false. The default value is false. If the value is true then Card details will be captured inside the CL Page.

Salt	String	This is a mandatory parameter that captures different elements mentioned below to create salt which is used for encryption. Following is an example of a salt JSON: {"txnId":"(MAY96c1d03348df4a8f92ad07019cb0ac39, MAY96c1d03348df4a8f92ad07019cb0ac40)", "txnAmount":"29.30", “deviceId”: ”ABCDEF”, “appId”:”com.psp1.app”, “mobileNumber”:”9002050725”, credType”:”reqBalEnq|pay|collect|setMpin|changeMpin|reqBalChk|mandate,,[pay, reqBalChk]/[collect,reqBalChk],“payerAddr”:”alex@sib”,“payeeAddr”:”rohan.patel@kbl”, “random”: “<Base64 String>” } Note: a. For “credType=pay|collect”, all the above mentioned fields are mandatory. Maintain unique txn_id. b. For “credType =reqBalEnq”, it will be a separate API call to UPI with unique txn_id. The payeeAddr must be same as payerAddr and “amount=0.00” c. For “credType =reqBalChk”, it will be a separate API call to UPI with unique txn-id. The payerAddr must be used as a payeeAddr. d. For the combination of pay&reqBalChk/collect&reqBalChk, credType will be formed as an array value. For e.g. “credType= [(pay, reqBalChk)/ (collect, reqBalChk)]”. e. In case of [Collect/reqBalCheck] ,QR or Intent where transaction ID is present in input parameters, Payer PSP needs to pass the original transaction id for generating collect/Pay cred block and for reqBalChk they have to generate new transaction ID. f. CL will form two cred block for the combination of PAY/COLLECT g. Base64 encoded 16byte random string generated by app, which is used as random for creating hash and also as iv parameter while AES encryption of hash for Trust.

Trust	String	a. Steps to generate Trust (mandatory field). Concatenate credType, transaction Id, app Id, mobile number, device id, payer address, payee address and transaction amount with the separator "|". b. 16-byte random value created. This random will be added in salt json as base64 string in random field. c. Create a hash of the concatenated string using SHA-256 algorithm with random value. d. Encrypt the hash with the token as key using AES-256 algorithm and random as iv parameter. Populate Trust with the encrypted string. Separate trust needs to be created for each credType and same has to be passed with the identified credType by forming a JSON String For example: Trust="{\"pay\":\"trust_for_type_pay\",\"reqBalChk\":\"trust_for_type_reqBalChk\"}"
payInfo	String	Sample of this JSON parameter is as below: { "name": "payeeName", "value": "Name of the Payee" }, { "name": "account", "value": "XXXXXXXX83647" }, { "name": "mobileNumber", "value": "919947496808" }, { "name": "txnAmount", "value": "235.45" }, { "name": "note", "value": "Pay for collect" }, { "name": "refId", "value": "1223423423" }, { "name": "refUrl", "value": "https://psp1.com" },

		{
		"name":"mandateSubType",
		"value":"create"
		}
		}
		The field mandateSubType is optional which is used for mandate related transaction and the possible values are “create”,” revoke”, “pause”, “unpause”, “modify”, “register”. For any other type app can ignore this value pair.

languagePref	String	en_US | hi_IN | gu_IN | or_IN | as_IN | bn_IN | kn_IN | te_IN | ml_IN | pa_IN | ta_IN | mr_IN | ur_IN |kok_IN |bho_IN |mni_IN |mwr_IN |kha_IN

5.4.2 Response

The response of the above will be a HashMap<String, String> (for Android) with the following attributes. There can be more than one entries in the HashMap, based on the number of credentials being captured by the common library.

The key of HashMap will have the credential name. This will use the name provided in the credentials. This should match with the “name” attribute in the control JSON in the input.

The value will be a JSON string containing the ki (key index) and the encrypted cred block. Example of the values in the HashMap is as per below. The keys of the Hash Map will be the subtype (e.g. MPIN)

Response for MPIN cred block

```
{
“MPIN “:{
"pay":  {
        "type": "PIN",
        "subtype": "MPIN",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
        } }
},
“det”: “2.2|<encrypted base 64 encoded data>"
}
```

Similarly an example of subtype: MPIN & ATMPIN cred block


```
{
“MPIN “:{
"setMpin":  {
        "type": "PIN",
        "subtype": "MPIN",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
        } }
},
"ATMPIN":{
    "setMpin":  {
        "type": "PIN",
        "subtype": "ATMPIN",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
        }}
},
“det”: “2.2|<encrypted base 64 encoded data>"
}
```


Similarly an example of subtype: card details cred block

Presently the last 6 digits of the Debit card number and expiry date are captured on the PSP App with base 64 encoding. In CL 1.7 version, we need to move the capturing of card details inside the NPCI Common library to ensure enhanced security .The same card detail cred block will be used in the ReqRegMob API for ”FORMAT3|FORMAT2” or “FORMAT3|FORMAT1” and Aadhar consent is not provided in app.


```
{

"CARDDETAILS “:{
  "setMpin":  {
        "type": "CARD",
        "subtype": "CARDDETAILS",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
```

```
          }}
"det": "2.2|<encrypted base 64 encoded data>"
}
```

Response for type: Balance Enquiry

```
{
"MPIN":{
"reqbalEnq": {
          "type": "PIN",
          "subtype": "MPIN",
          "data": {
                    "code": "NPCI",
                    "ki": "20150822",
                    "encryptedBase64String": "2.2|<encrypted base 64 encoded authentication data>"
          }} ,
"det": "2.2|<encrypted base 64 encoded data>"
}
```

Response for type: Balance Check

```
{
     "reqbalChk": {
            {
          "type": "PIN",
          "subtype": "MPIN",
          "data": {
                    "code": "NPCI",
                    "ki": "20150822",
                    "encryptedBase64String": "2.2|<encrypted base 64 encoded authentication data>"
          } }
},
"det": "2.2|<encrypted base 64 encoded data>"
}
```

Response for type: Pay

```
{
"MPIN": {
"pay": {
          "type": "PIN",
          "subtype": "MPIN",
          "data": {
                    "code": "NPCI",
                    "ki": "20150822",
                    "encryptedBase64String": "2.2|<encrypted base 64 encoded authentication data>"
          }} ,
```

```
"det": "2.2|<encrypted base 64 encoded data>"
}
```

Response for type: Collect

```
{
"MPIN": {
"collect":  {
        "type": "PIN",
        "subtype": "MPIN",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
        } },
"det": "2.2|<encrypted base 64 encoded data>"
}
```

Response for type: mandate

```
{
"MPIN": {
"mandate": {
        "type": "PIN",
        "subtype": "MPIN",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
        } },
"det": "2.2|<encrypted base 64 encoded data>"
}
```

Response for type: (PAY, ReqBalChk)| (COLLECT, ReqBalChk)

```
{
"MPIN": {
  "pay":  {
        "type": "PIN",
        "subtype": "MPIN",
        "data": {
                "code": "NPCI",
                "ki": "20150822",
                "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
        } }
}

"MPIN": {
```

```
    "reqBalChk": {
        "type": "PIN",
        "subtype": "MPIN",
        "data": {
            "code": "NPCI",
            "ki": "20150822",
            "encryptedBase64String": "2.2|<encrypted base 64 encoded authentication data>"
        } }
},
"det": "2.2|<encrypted base 64 encoded data>"
}
```


In case there is an error in common library, the returned JSON string will be as below:

```
"error": { "errorCode" : <errorCode>, : "errorText" : <errorText> }
```

For example Digital Signature mismatch in common library, the error will be as below:

```
"error": { "errorCode" : "L05", : "errorText" : "Technical Issue, please contact your administrator" }
```

6 Rotation of Token

The token needs to be refreshed every month, the PSP mobile app should have option to rotate the token. If token expires the common library would fail to capture credential if requested. For rotating or resetting the token, the following steps need to be executed sequentially.

1. Get Challenge: Generate the "*Get Challenge*" service with type = "**rotate**" to receive a challenge from common library.
2. Get Token: Pass the generated challenge to PSP server to execute the "Get Token" service of UPI to receive a valid token. This token should be stored at PSP mobile app end to ensure secure communication with common library.
   The objective is to get the "token" from UPI. For that we have a placeholder in the sample code, which does not require any interface with the NPCI library. The token is obtained by passing the "challenge" as an input parameter to the ReqListKeys API. The API is as below, and the necessary fields for "*Get Token*" has been highlighted .

```
<upi:ReqListKeys xmlns:upi="http://npci.org/upi/schema/">

<Head ver="1.0" ts="" orgId="" msgId=""/>

<Txn id="" note="" refId="" refUrl="" ts="" type="ListKeys/GetToken"/>
```

```
        <Creds>
                <Cred type=" Challenge" subtype=" initial/rotate">
                <data code=""
                ki="">device id|app id|mobile number|challenge</data>
                //Note: device id, app id, mobile number and challenge are
                concatenated including pipe characters in between.
                </Cred>
        </Creds>
</upi:ReqListKeys>
```

The RespListKeys returns the following:

```
<key code="NPCI" type="PKI" ki="yyyymmdd">
   <keyValue>base64 encoded token</keyValue>
</key>
```

Here the highlighted part is the token in a Base64 encoded format. This can be decoded to Byte Array and encoded again as Hex String. For all subsequent communication to NPCI Library, this Hex String is to be used as a token.

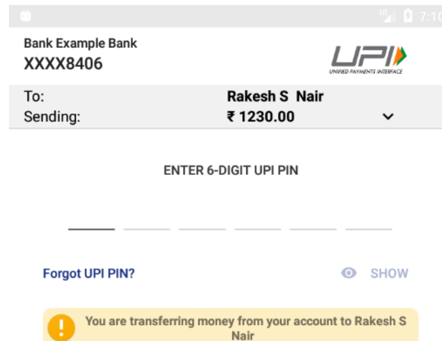
3. Register App: Execute the "Register App" service to verify and register the token in the Common Library. Signature of the required services have already been described in the Registration with the Common library (5.3) section. Sample code for integration available in the Annexure-1.

7 Get Credential service (User interface)

The common library is used to capture the credentials for the following cases:

a. Debit authorization
b. Balance Enquiry
c. Balance Check
d. Set/Forgot/Change UPI PIN
e. Mobile Banking Registration

For Debit authorization, the PSP is requested to provide all the parameters in the JSON request. The UPI user screen will look as below:



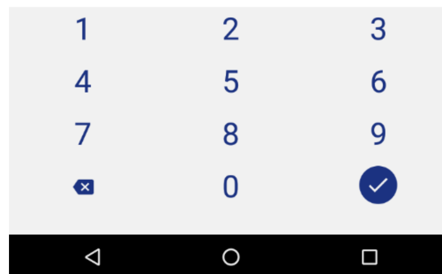


Figure: Debit Authorization

UNIFIED PAYMENTS INTERFACE

For Balance Enquiry, the PSP can use txnAmount=0.00, payerAddr, payeeAddr, payeeName, refId, refUrl, note (all payer details will be used as payee details) from the payInfo JSON in the Salt, but account number should be provided in the account tag. The NPCI Library will look as below:

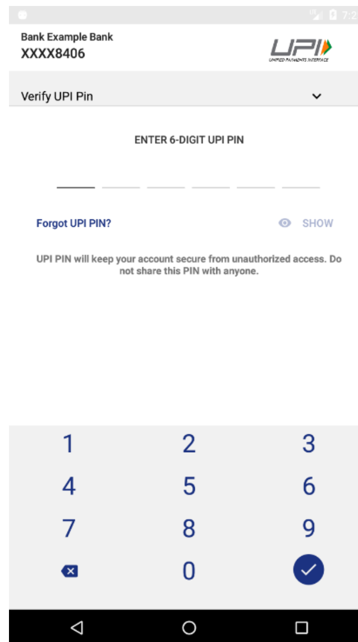


Figure: Balance Inquiry

For Balance Check, the PSP is requested to provide all the parameters, PSP can use txnAmount=0.00, payerAddr, payeeAddr, payeeName, refId, refUrl, note (payer details will be used as payee details) from the JSON request. According to the response from the issuer bank, payer psp app should process the request. The UPI user screen will look as below:

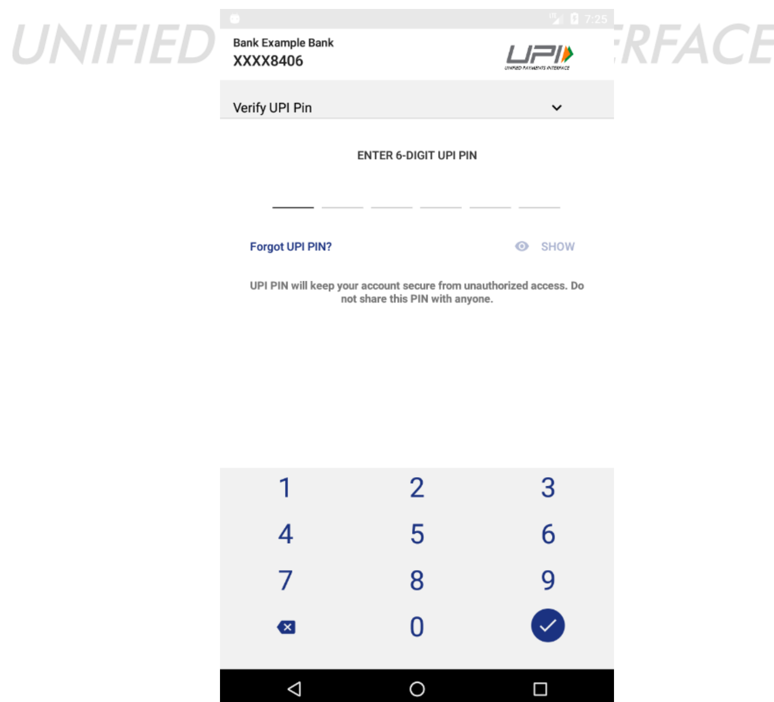


Figure: Balance Check

For Change UPI PIN, the PSP JSON request as like below to capture the old and new UPI PIN. Also app should sent credType as "changeMpin" to CL.


```
{
  "CredAllowed": [{
        "type": "PIN",
        "subtype": "MPIN",
        "dtype": "NUM | ALPH",
        "dlength": "6"
  }, {
        "type": " PIN ",
        "subtype": "NMPIN ",
        "dType": "NUM | ALPH",
        "dLength": "6"
  }]}
```


While sending the message to UPI, PSP has to change the NMPIN above to MPIN, otherwise there will be validation error in UPI.

The PSP can skip txnAmount, payerAddr, payeeAddr, payeeName, refId, refUrl, note, account, from the payInfo JSON in the Salt JSON request. The NPCI Library will look as below:

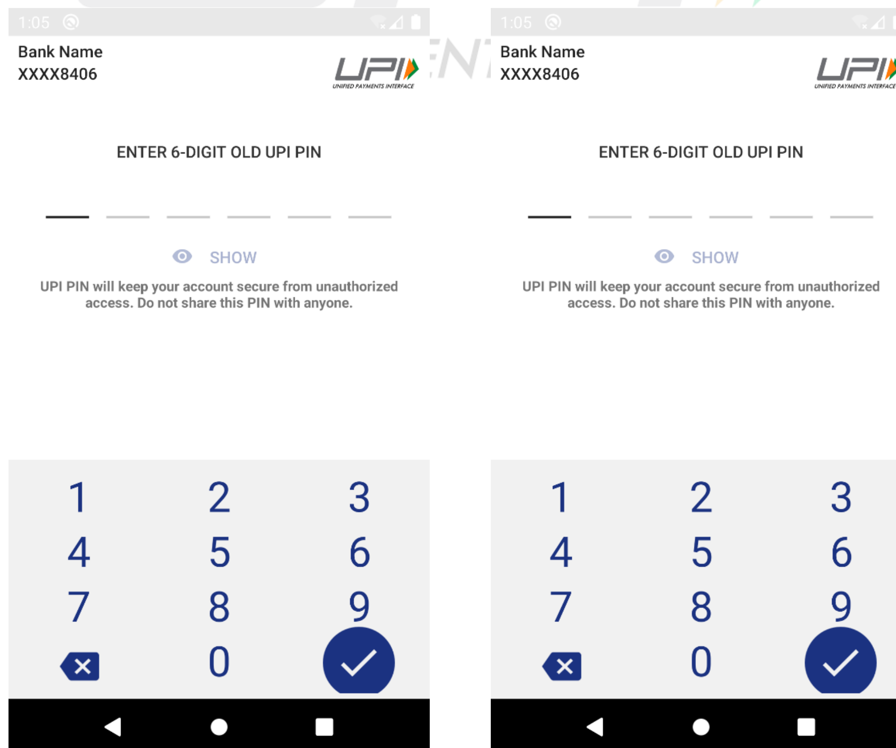


Figure: Change UPI PIN

For Mobile banking registration and set/forgot UPI PIN, the PSP JSON request as like below to capture the OTP & new UPI PIN. The below sample is for FORMAT1 if it is FORMAT2 please add ATMPIN cred block also.


```
{
        "CredAllowed": [{
                "type": "PIN",
                "subtype": "MPIN",
                "dtype": "NUM | ALPH",
                "dlength": "6"
        }, {
                "type": "OTP",
                "subtype": "SMS",
                "dType": "NUM | ALPH",
                "dLength": "6"}]        }
```


The PSP can skip txnAmount, payerAddr, payeeAddr, payeeName, refId, refUrl, note, account, from the payInfo JSON in the Salt JSON request. The NPCI Library will look as below:

Figure: Mobile Bank Registration

8 New feature of UPI CL 1.5

8.1 OTP Auto detection

New library can auto read the OTP from issuer SMS message. PSP app should request the permission from device for SMS_READ and SMS_RECEIVE. (Declaring this in manifest or taking at runtime in case of Android Marshmallow or above)

Issuer bank need to follow any one of the standard OTP message to facilitate auto read as mentioned below

OTP is xxxxxx

One Time Password is xxxxxx

xxxxxx is your OTP

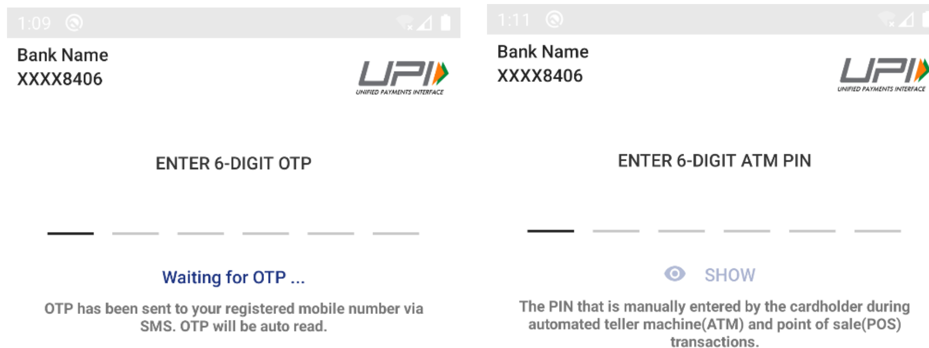
xxxxxx is your one time password

8.2 ATMPIN + OTP Flow in Registration

This version of library also supports screen for ATMPIN flow where user shall enter OTP along with ATMPIN and then the new MPIN he is trying to set. In this case, PSP apps shall pass the JSON as follows for FORMAT2 (for the FORMAT1 it can avoid ATMPIN data values):


```
{
  "CredAllowed": [
    {
      "type": "OTP",
      "subtype": "SMS",
      "dType": "NUM",
      "dLength": "6"
    },
    {
      "type": "PIN",
      "subtype": "ATMPIN",
      "dType": "NUM",
      "dLength": "4"
    },
    {
      "type": "PIN",
      "subtype": "MPIN",
      "dType": "NUM",
      "dLength": "6"
    }
  ]
}
```


The PSP can skip txnAmount, payerAddr, payeeAddr, payeeName, refId, refUrl, note, account, from the payInfo JSON in the Salt JSON request. The NPCI Library will look as below:



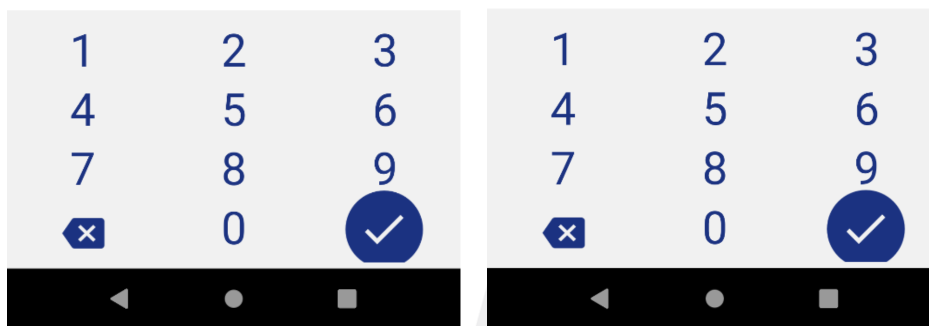


Figure: Mobile banking registration using ATM PIN

8.3 Virtual/Custom Numeric Keyboard

To save user's UPI PIN from malicious attack like Keylogging, common library now uses its own numeric keyboard.

9 New feature of UPI CL 1.7

9.1 Multilingual Support

Now common library supports following languages: Hindi (hi_IN), English (en_Us), Gujarati (gu_IN), Oriya (or_IN), Assamese (as_IN), Bengali (bn_IN), Kannada (kn_IN), Telugu (te_IN), Malayalam (ml_IN), Punjabi (pa_IN), Tamil (ta_IN) , Marathi (ma_IN), Marwari(mwr_IN), Manipuri(mni_IN), Khasi(kha_IN), Bhojpuri(bho_IN), and Konkini(kok_IN). While calling "getCredential", PSP app should pass the chosen language preferences as a parameter. Please pass the code for "languagePref"as mentioned in brackets.

9.2 Verified Merchant Confirmation

To display whether the merchant is verified in the details bar, PSP app has to pass a parameter called "verifiedMerchant" as follow:

```
{
    "payerBankName":"Indian Bank Ltd.",
    "backgroundColor": "#FF9933",
    “color”: "#FF9933",
    ”verifiedMerchant”:”true”
}
```

9.3 Resend OTP

PSP app can use the resend OTP, in case there is issue in sending or receiving from issuer side, hence a button is available in the library page after 60 secs to request for OTP in case of setting UPI PIN flow.
Also, the PSP app can specify the number of attempts the user can request for resend OTP feature. By default, the number of attempts will be 2 times.
"issuerResendOTPLimit" will be used for issuer resend OTP count and "aadhaarResendOTPLimit" used for Aadhar resend OTP Count,

To do this, PSP app has to pass parameters called "*resendAadharOTPFeature*" and "*resendIssuerOTPFeature*" as true in configuration:

```
{
    "payerBankName":"Indian Bank Ltd.",
    "backgroundColor": "#FF9933",
    “color”: "#FF9933",
    "resendAadhaarOTPFeature": "true",
    "resendIssuerOTPFeature": "true",
     "issuerResendOTPLimit": "2",
    "aadhaarResendOTPLimit": "2",
}
```

Along with this, PSP app should take the resultCode = 2 (for issuer OTP) and resultCode=4(for Aadhaar OTP) which will be returned in case of “*Resend OTP*” button

clicked on NPCI common library page. On receiving the result code, PSP app should trigger the OTP again in case of registration/ setting UPI PIN flow
Once OTP has been generated by PSP same needs to be communicated back to CL
To obtain this, execute() function on CL needs to be called with the following data by PSP or merchant app.

Expected data is a JSON in the format

```
{
"type":"TRIGGERED_OTP_RESPONSE|TRIGGERED_AADHAAR_OTP_RESPONSE",
"data":{
"status":"0",

}
}
```

Parameter Name	Data Type	Description
Status	String	This is a mandatory parameter specifying status of otp generation - "0" for otp generation success, "-1" for otp generation failure.

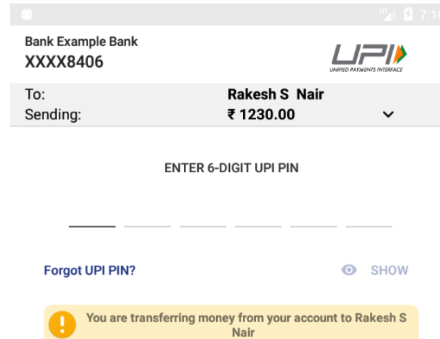
9.4 Forgot UPI PIN

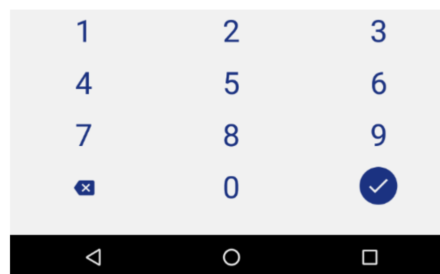
PSP app can use the forgot UPI PIN button to get notified, in case the user has forgotten the UPI PIN and wish to change it when requesting a transaction or in the middle of the transaction.

To do this, PSP app has to pass a parameter called "forgotUpiPINEnabled" in configuration as follows:

```
{
    "payerBankName":"Indian Bank Ltd.",
    "backgroundColor": "#FF9933",
    "color": "#FF9933",
    ”forgotUpiPINEnabled”:”true”
}
```

Once the “forgotUpiPINEnabled” configuration is passed with the boolean value “true”, the NPCI Common Library Page will display the “Forgot UPI PIN” button as follow:





Along with this, PSP app should take the resultCode = 3 which will be returned in case of “Forgot UPI PIN” button clicked on NPCI common library page. On receiving the result code, PSP app should trigger the Reset UPI PIN flow.

9.5 Capturing of debit card details and expiry date on CL page

Presently the last 6 digits of the Debit card number and expiry date are captured on the PSP App with base 64 encoding. In CL 1.7 version, we need to move the capturing of the card details inside the NPCI Common library to ensure enhanced security .CL will provide cred block with the card details to the App. So the PSP need to send this same cred in the ReqRegMob API.

Sample Cred output from CL to app:

```
{
"CARDDETAILS ":{
  "setMpin":  {
"type": "CARD",
          "subtype": "CARDDETAILS",
          "data": {
"code": "NPCI",
"ki": "20150822",
                  "encryptedBase64String":   "2.2|<encrypted   base   64   encoded
authentication data>"
```

}}
"det": "2.2|<encrypted base 64 encoded data>"
}

9.5.1 Set/Change UPI Pin

Starting Common Library 1.7, debit card flow has been moved inside common library based on the app input, to provide a uniform and secure environment to user.

From CL 1.7 and above, for Issuer banks on either FORMAT3|FORMAT1 or FORMAT3|FORMAT2, user will be required to input the card details on CL Screen (when Aadhaar Consent not provided). In case of Aadhaar consent provided, Card details will not be fetched.

CL will generate a Cred Block with these card details entered by the user and the same will be passed in ReqRegMob API to UPI.

Sample output shown below(SetMpin+Atm):

```
{
    "credBlocks":    {
        "ATMPIN":       {
            "SetMpin":          {
                "data":             {
                    "code": "NPCI",
                    "encryptedBase64String": "2.2|<encrypted authentication data>"",
                    "ki": "20150822"
                },
                "subType": "ATMPIN",
                "type": "PIN"
            }
        },
        "DEBIT":        {
            "SetMpin":          {
                "data":             {
                    "code": "NPCI",
                    "encryptedBase64String": "2.2|<encrypted authentication data>",
                    "ki": "20150822"
```

```
                },
                "subType": "CARDDETAILS",
                "type": "CARD"
            }
        },
        "MPIN":        {
            "SetMpin":           {
                "data":               {
                    "code": "NPCI",
                    "encryptedBase64String": "2.2|<encrypted authentication data>",
                    "ki": "20150822"
                },
                "subType": "MPIN",
                "type": "PIN"
            }
        },
        "SMS":        {
            "SetMpin":           {
                "data":               {
                    "code": "NPCI",
                    "encryptedBase64String": "2.2|<encrypted authentication data>",
                    "ki": "20150822"
                }
                "subType": "SMS",
                "type": "OTP"
            }
        },
}
    }
                    "det": "2.2|<encrypted authentication data>",
}
```


Sample output shown below(SetMpin+Aadhaar OTP):


```
{
    "credBlocks":    {
```

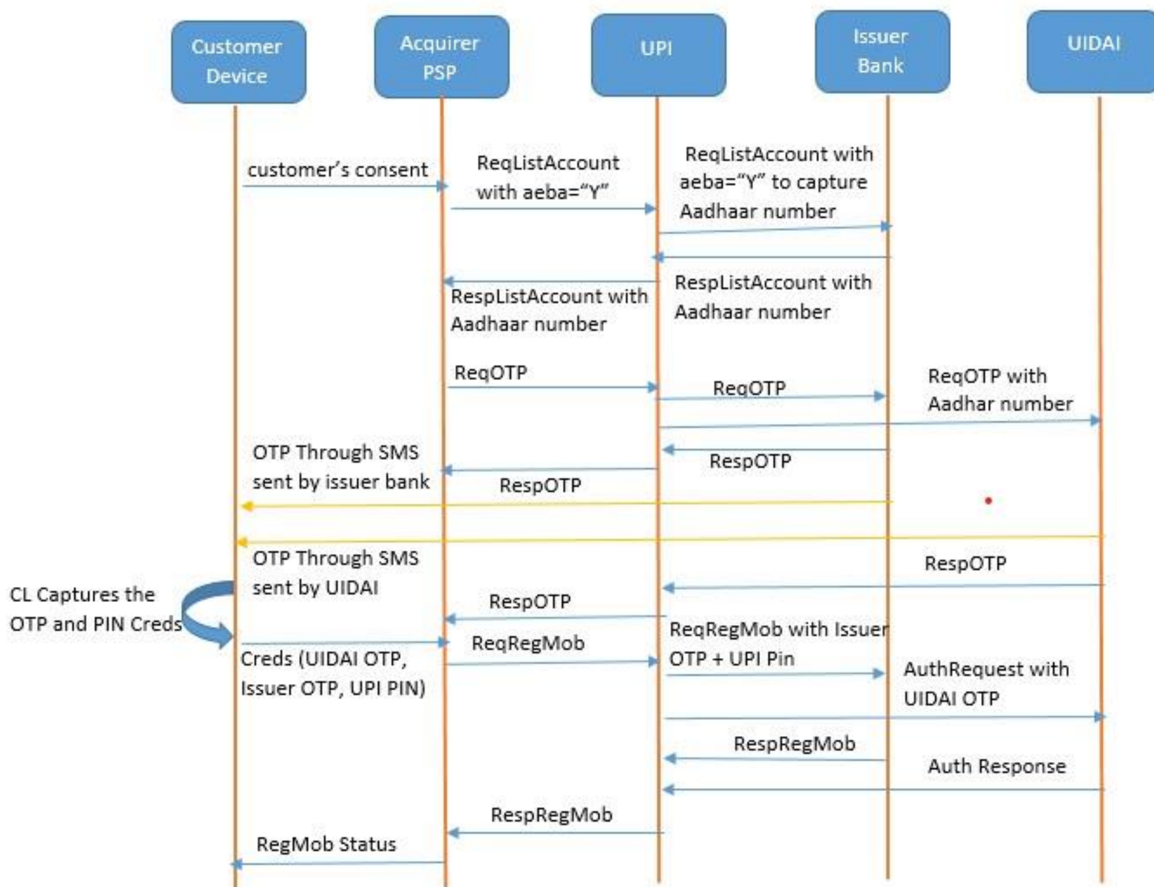
```
        "MPIN":         {
            "SetMpin":            {
                "data":             {
                    "code": "NPCI",
                    "encryptedBase64String": "2.2|<encrypted authentication data>",
                    "ki": "20150822"
                },
                "subType": "MPIN",
                "type": "PIN"
            }
        },
        "SMS":        {
            "SetMpin":            {
                "data":             {
                    "code": "NPCI",
                    "encryptedBase64String": "2.2|<encrypted authentication data>",
                    "ki": "20150822"
                }
                "subType": "SMS",
                "type": "OTP"
            }
        },
     {
" AADHAAR“:{
"setMpin": {
"type": "OTP",
"subtype": "AADHAAR"
"data": {
"code": "NPCI",
"ki": "20150822",
                    "encryptedBase64String":    "2.2|<encrypted    base    64    encoded
authentication data>"
         }
}
}
    }
                    "det": "2.2|<encrypted authentication data>",
```

```
}
```

9.5.2 Aadhaar OTP Authentication

Aadhaar number authentication can be used as an authentication method at the time of user onboarding.

Sample Flow Diagram:



The following steps will be performed for Aadhaar OTP Authentication.

1. *ResqListAccount API call with Aadhaarconsent="Y"*
2. *CL control will have Type: OTP, subtype: Aadhaar, ReqOTP trigger both to Issuer Bank and UIDAI and receive two OTPs.*
3. *To capture the required Information CL need to be triggered with the following controls:*

```
{
"CredAllowed": [
```

```
    {
     "type": "OTP",
      "subtype": "SMS",
      "dType": "NUM",
      "dLength": "6"
    },

    {
      "type": "PIN",
      "subtype": "MPIN",
      "dType": "NUM",
      "dLength": "6"
    }
   {
      "type": "OTP",
      "subtype": "AADHAAR",
      "dType": "NUM",
      "dLength": "6"
     },
}
  ]
}
```


4. RegRegMob will have UIDAI OTP+ISS OTP+UPIPIN. Send UIDAI OTP to UIDAI and Validate with UIDAI .
5. If UIDAI validation success, call issuer bank with ISS OTP+UPIPIN and send FinalResponse to Payer PSP

For Further Details, Please Check Unified Payments Interface 2 0 TSD v1 42

9.6 Capturing of device details.

In CL 1.7 for additional security some device details are captured and which will be encrypted and shared along with each of the cred blocks. The list of device parameters adding to CL output are attached.

CL Device Parameters - v0.1.xls

The output from the CL after capturing the device parameters is:

Response :


```
{
    "credBlocks":    {
```

```
"ATMPIN":        {
    "setMpin":            {
        "data":            {
            "code": "NPCI",
            "encryptedBase64String": "2.2|<encrypted authentication data>"",
            "ki": "20150822"
        },
        "subType": "ATMPIN",
        "type": "PIN"
    }
},
"CARDDETAILS":        {
    "setMpin":            {
        "data":            {
            "code": "NPCI",
            "encryptedBase64String": "2.2|<encrypted authentication data>",
            "ki": "20150822"
        },
        "subType": "CARDDETAILS",
        "type": "CARD"
    }
},
"MPIN":        {
    "setMpin":            {
        "data":            {
            "code": "NPCI",
            "encryptedBase64String": "2.2|<encrypted authentication data>",
            "ki": "20150822"
        },
        "subType": "MPIN",
        "type": "PIN"
    }
},
"SMS":        {
    "setMpin":            {
        "data":            {
            "code": "NPCI",
            "encryptedBase64String": "2.2|<encrypted authentication data>",
            "ki": "20150822"
        }
        "subType": "SMS",
        "type": "OTP"
```

```
            }
        }
    }
                    "det": "2.2|<encrypted authentication data>",
}
```

The encrypted value will be added in to the output as a new value as “det”. PSP has to send this value to the UPI along with cred block as a new attribute to the creds.

Sample Cred to be shared:

```
  <Creds ki=”2.2|<encrypted base 64 encoded data”>
<Cred type="OTP" subType="AADHAAR"> <Datacode="" Ki=""> base-64 encoded/
encrypted authentication data </Data> </Cred>
 <Cred type="OTP" subtype="SMS|EMAIL|HOTP|TOTP">
      <Data code="NPCI" ki="20150822> Bae64 Encoded/Encrypted data from NPCI
common library</Data>
    </Cred>
    <Cred type="PIN" subtype="MPIN">
      <Data code="NPCI" ki="20150822"> Bae64 Encoded/Encrypted data from NPCI
common library</Data>
    </Cred>
    <Cred type="CARD" subType="CARDDETAILS">
       <Data code="NPCI" ki="20150822"> Bae64 Encoded/Encrypted data from NPCI
common library</Data>
    </Cred>
</Creds>
```

The attribute “ki” contains the encrypted device details. UPI will validate this data for security check

10. Encryption Algorithm

The cred block data will be base 64 encoded. RSA 2048 public key encryption will be used to encrypt the credential block for credentials to be sent to UPI

11. Common Library Process Flow

1) PSP mobile application would invoke the common library using the defined interface
2) Common library would first validate the digitally signed xml payload of public keys. If validation is unsuccessful, it will return error accordingly.
3) Common library would pick any one key from the list of keys provided as input.

4) Depending upon the input, credential(s) is/are to be captured, common library will display a UI to the user. Basic information about the transaction will be displayed. (Passed from PSP).
5) Common library would encrypt the data using UPI public key and return a Base64 string.
6) Common library would return the response to PSP mobile Application.

12. Generation of XML payload in PSP server

Once the PSP server gets the encrypted credentials from the PSP app, it generates the Cred block in the following manner.

```
  <Creds ki="2.2|<encrypted base 64 encoded data">
<! – The below block will be used if the payer psp is getting the card details in the CL page – > <Cred type="CARD" subType="CARDDETAILS">   <Data code="" ki="">
base-64 encoded/encrypted authentication data </Data> </Cred>    <! Consists of MOBILE, CARD DIGITS, EXPDATE!> <! This cred block is used when the payer psp has upgraded to new CL version, which supports capture of card detail in CL itself–>
    <Cred type="OTP" subtype="SMS|EMAIL|HOTP|TOTP">
        <Data code="NPCI" ki="20150822> Bae64 Encoded/Encrypted data from NPCI common library</Data>
     </Cred>
<Cred type="PIN" subtype="ATMPIN">
        <Data code="NPCI" ki="20150822"> Bae64 Encoded/Encrypted data from NPCI common library</Data>
     </Cred>
     <Cred type="PIN" subtype="MPIN">
        <Data code="NPCI" ki="20150822"> Bae64 Encoded/Encrypted data from NPCI common library</Data>
     </Cred>
<Cred type="OTP" subType="AADHAAR"> <Datacode="" Ki=""> base-64 encoded/ encrypted authentication data </Data> </Cred>
     </Cred>
</Creds>
```

13. Versioning Strategy

Common library releases will maintain a version number for different platforms. The UPI server would store all the valid version numbers in the server. The version number of common library would reside in the encrypted block. At UPI end the same would be validated.

14. Configuration Parameters in Common Library

1. Configuration options will be there to maintain size of the MPIN/OTP for banks.

2. Configuration is to be maintained for the credentials which UI component will displayed. For example, for MPIN entry a text will be displayed. When the consumer tries to set MPIN, last 6 digits of debit card, OTP and new PIN are to be entered and it will be captured. In this scenario, these three fields are to be displayed by the common library. Since last six digits of card number and expiry date will be encrypted, PSPs are suggested not to capture these two fields outside the NPCI Library.
3. The structure of the credential block may vary based on common library version. This configuration is to be maintained and validated in UPI.

15. Common library release and distribution

Common library would be distributed as following:

1) Signed AAR.jar file in case of android

UPI would publish the common library to a specific location to be downloaded by the PSP. A new version of common library release would be needed in-case of any change in the following: encryption logic / biometric vendor's SPI defined by NPCI / UI to capture data, bug fixes, performance fixes, changes to underlying OS versions, etc.

16. In-app payment/Deep-linking from merchant apps

This is a scenario where merchant app would use the UPI payments system through a PSP app. This can be done in two ways:

a) A collect request that merchant app would initiate through their server through its PSP (merchant being enrolled with the PSP). The end user will respond to the collect call and pay.
b) A consumer uses a merchant app for payment. The merchant app would invoke implicit intent for UPI Payment enabled app. UPI payment enabled app would expose their intent and thus android will show the users option(s) of UPI payment enabled app to be used (based on number of such apps present in the device). The merchant app should pass the transaction amount to the PSP app. PSP app will now invoke the NPCI common library to capture the credentials. The end user will review and initiate the payment. The PSP app will communicate with the PSP API gateway, which will subsequently send a ReqPay to UPI along with the cred block. PSP app will communicate the response to merchant app.

In case of android this will be achieved by IMPLICIT Intent. Merchant app would make an Intent request and the PSP app installed in users mobile would respond to that intent. User will review the submission screen and submit the payment.

18.1 Recommendation on parameters

Please refer to COMMON URL SPECIFICATIONS DEEP LINKING AND PROXIMITY INTEGRATION document version 1.0 for recommendation on parameters that should be passed between the merchant and PSP/Bank app.

17. Deep-linking configuration guidelines for Android

PSP should configure their app the configuration procedure is elaborated below.

1. Android Manifest

```
<activity android:name=".PayActivity" android:label="@string/app_name">
        <intent-filter android:label="@string/pay">
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <category android:name="android.intent.category.BROWSABLE"/>
                <!–Accepts URIs that begin with "upi://pay”–>
                <data android:scheme="upi" android:host="pay"/>
        </intent-filter>
</activity>
```

In the above configuration, “Pay Activity” is the class name of the activity which is invoked for the payment. The **data** tag is needed to form the URL which the merchant app needs to execute to initiate the payment. Each and every PSP app should have the same URL so that when the merchant app invokes the payment, the user (payer) would be able to find a list of all available PSP apps in the device.

2. Pay Activity

When PSP app receives call from merchant app, it would need to extract different parameters sent by the merchant app and create transaction request object which contains the information required to initiate a transaction. It should do it in the Pay Activity.

```
private TransactionRequest request = null;
@Override
protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent deepLinkingIntent= getIntent();
        //Receive parameters from merchant if called from app
        request = processRequestIntent(deepLinkingIntent);
}
```

After processing the transaction, PSP app needs to send the response to the merchant app. Procedure is provided below.

```
Intent responseintent = createResponseIntent();
```

```
//Populate response fields which need to be sent to merchant app
Activity.setResult(0, responseIntent);
```

3. PSP URL

Merchant app needs to create the URL and execute the PSP app using the below procedure.

```
protected static final String DEEPLINKING_URL_BASE = "upi://pay";
// populate url with the parameters according to specification
StringBuilder urlBuilder = new StringBuilder();
urlBuilder.append(DEEP_LINK_URL_BASE).append("?")
            .append(pa).append ("=").append(payeeVpa).append("&")
            .append(pn).append ("=").append(payeeVpa).append("&")
            .append(mc).append ("=").append(payeeMcc).append("&")
            .append(ti).append ("=").append(txnId).append("&")
            .append(tr).append ("=").append(txnRef).append("&")
            .append(tn).append ("=").append(txnNote).append("&")
            .append(am).append ("=").append(payeeAmt).append("&")
            .append(cu).append ("=").append(payeeCur).append("&")
            .append(appid).append ("=").append(payeeAppId).append("&")
            .append(appname).append ("=").append(payeeAppName);
String deepLinkUrl = urlBuilder.toString();
```

4. On click listener of “Pay by UPI (using same mobile)” button

```
Intent intent = new Intent(Intent.ACTION_VIEW);
Intent.setData(Uri.parse(deepLinkUrl));
String title = "Pay with";
//Create intent to show chooser. It will display the list of available PSP apps (which have the same url in the manifest)
Intent chooser = Intent.createChooser(intent, title);
//Verify the intent will resolve to at least one activity
If (intent.resolveActivity(getPackageManager()) !=null) {
            startActivityForResult(chooser, 1);
}
```

When user (payer) clicks on the “Pay by UPI (Using same mobile)” button, the merchant app would create a list of PSPs available in the device which can make the payment. User chooses one of the PSP apps to make the payment. Merchant app invokes the PSP App and expects a response from it.

Merchant app should process the response using the below procedure.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent responseIntent){
        super.onActivityResult(requestCode, resultCode, responseIntent);
        if (requestCode == 1 ) {
```

```
                // extract the response code and other fields from the received intent
                processResponseIntent(responseIntent);
        }
}
```

18. List of FAQs

a) **What is TOKEN?**

After mobile number verification, PSP app should call "Get Challenge" and then call API for "GetToken". As a result of this, NPCI creates a token associated to appId, device and mobile number. This token should be stored in PSP mobile app to ensure secure communication with Common Library

b) **What is the rotation period of TOKEN?**

Token has an expiry of 30 days. So when user tries for a transaction, PSP should check the creation date of token and if expired, it shall make a rotation call and renew the token and replace it with older one.

c) **What is the min SDK version supports & required runtime permissions?**

Common Library supports API level 23 as min SDK version.

As per new features of v1.5, it can auto detect the OTP and for this it requires permission for READ_SMS and RECEIVE_SMS. PSP apps can declare this in their manifest and to handle permissions above Android Marshmallow, they shall take runtime permission for SMS group from user before opening common library (in case of OTP as a part of credBlock)

d) **What are the steps need to be followed in case of "Technical Error" while invoking Common Library?**

i. Recommend to check all seven arguments used in "getCredential" call
ii. All the arguments are mandatory
iii. Check the JSON objects structure
iv. Verify the logs which contains CLException & reason will be displayed in logcat for developers

e) **What is the action required while getting "U63" in during Device Registration?**

After calling "getChallenge", make sure the same device Id is passed in API which was used while calling "getChallenge". Also check if you are using correct version of aar and jar file as per environment. For an example, if UAT libraries are used in PROD environment, it may throw this error.

f) **What is the reason for getting "U66" in pay request or other credBlock calls?**

Make sure that the token created with combination of app Id, device Id, mobile number is the one which was called with "Register App" throwing out true value. This Register App call is mandatory to verify if token received from the server is valid or not.

We also recommend to check if the amount passed in case of PAY REQUEST is in two decimals like "2.00" and not "2".

g) **How to change common library background color?**

From version 1.5 of common library, there will be no color customizations for banks on CL page. So you can ignore configurations for colors from v1.5.

19. Contact Us

For further queries, request to drop mail to upi.helpdesk@npci.org.in

20. Annexure-1

21.1 MainActivity.java

```
package Sample;
public class MainActivity extends Activity {

    public String TOKEN;
    public String salt;
    public String trust;
    public String random;
    public String payInfoArray;

    boolean result;
    String keyCode = "NPCI";
    String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?><ns2:RespListKeys
xmlns:ns2=\"http://npci.org/upi/schema/\"
xmlns:ns3=\"http://npci.org/cm/schema/\"><Head
msgId=\"1GRDBknspOwTP5TLnjQ6\" orgId=\"NPCI\" ts=\"2018-01-
31T15:20:20+05:30\" ver=\"1.0\"/><Resp
reqMsgId=\"SBI8a649e16308642f3a39408a0d42fe940\" result=\"SUCCESS\"/><Txn
id=\"SBI170314CCB08F41BAA6C333C1D8771223\" note=\"ListKeys\"
refId=\"SBI170314CCB08F41BAA6C333C1D8771223\"
refUrl=\"https://www.sbi.co.in\" ts=\"2018-01-31T15:23:24+05:30\"
type=\"ListKeys\"/><keyList><key code=\"NPCI\" ki=\"20150822\" owner=\"NPCI\"
type=\"PKI\"><keyValue xmlns:xs=\"http://www.w3.org/2001/XMLSchema\"
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xsi:type=\"xs:string\">MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAu
MKxWfy0WcPp98muBWa6yhpmb6ZGZGSKHRIOv05UIIN5TbUPl6yEerh7Wj0+JyKf
sOntRdAVhkLJGRoHwH6gEEeFNHge7kPea/B33cQAbqa39mnP5F1aaZT3tjJnKrfI1
Wum0crdb7dAMzft4JILOEa+s3Uh7OdYEI/Xp7EisdSoJ345Cj0LTfLZEQzRdVGovXZr
```

```
fLByJysH11V9tDrIVv75C/3UndwjHt3NrqzNBoUMh5VZRFkcwuebUAkhIed5gvoysJ
wd0yYGrAUXNrXJJDTAj5diCuasWyfWZR9IsX5I14hdxF+IqadR/pgII53DW5oEy2LM
Xgvt2u/qmSml8wIDAQAB</keyValue></key></keyList><Signature
xmlns=\"http://www.w3.org/2000/09/xmldsig#\"><SignedInfo><Canonicalization
Method Algorithm=\"http://www.w3.org/TR/2001/REC-xml-c14n-
20010315\"/><SignatureMethod
Algorithm=\"http://www.w3.org/2001/04/xmldsig-more#rsa-
sha256\"/><Reference URI=\"\"><Transforms><Transform
Algorithm=\"http://www.w3.org/2000/09/xmldsig#enveloped-
signature\"/></Transforms><DigestMethod
Algorithm=\"http://www.w3.org/2001/04/xmlenc#sha256\"/><DigestValue>tZnN
xFfuFmXB9jHhZAlVnEG2/sfwWVX97ZSHLbm1s6Y=</DigestValue></Reference></
SignedInfo><SignatureValue>Sq8jDGHAi1Z0GiDgpnDWmtsfQT/mSN+wQ/adNXB
eafzRbWa05CVlyW7ZCzRfBTcTdVbE7HDdsUE4\n" +
          "
LZehJAYP6uxXTvb28Tx0gf6IRY5OQAd9XoS3kdo22kr9vXQ22iV8GkHlyC6ouuq8D
OQCKMxquG5W\n" +
          "
09UyTwOVYt2uIMACgJKykMeaP6uma7/9D3ewuBDo8GIsYszzQRxHocGLxO9+miL
NIz+hNgtmBYSP\n" +
          "
BW/MCgquL+QYBegY3Q6J4C4mSqm3jr0Yd+A9bX8m/STf19zaoGPh7xukpdLeO0
n3EDS66I9AVrBz\n" +
          "
OooDvFmDsiaFPT7zLoWWzWfH1ndSVEWb0R+84g==</SignatureValue><KeyInfo
><KeyValue><RSAKeyValue><Modulus>uMKxWfy0WcPp98muBWa6yhpmb6ZGZG
SKHRIOv05UIIN5TbUPl6yEerh7Wj0+JyKfsOntRdAVhkLJ\n" +
          "
GRoHwH6gEEeFNHge7kPea/B33cQAbqa39mnP5F1aaZT3tjJnKrfI1Wum0crdb7d
AMzft4JILOEa+\n" +
          "
s3Uh7OdYEl/Xp7EisdSoJ345Cj0LTfLZEQzRdVGovXZrfLByJysH11V9tDrIVv75C/3U
ndwjHt3N\n" +
          "
rqzNBoUMh5VZRFkcwuebUAkhIed5gvoysJwd0yYGrAUXNrXJJDTAj5diCuasWyfW
ZR9IsX5I14hd\n" +
          "
xF+IqadR/pgII53DW5oEy2LMXgvt2u/qmSml8w==</Modulus><Exponent>AQAB</
Exponent></RSAKeyValue></KeyValue></KeyInfo></Signature></ns2:RespListKeys
>";

    String languagePref = "";
    @Override
    protected void onCreate(Bundle savedInstanceState) {

        languagePref = language;

        if(Constant.clServices == null){
            CLServices.initService(this, new ServiceConnectionStatusNotifier() {
```

```
            @Override
            public void serviceConnected(CLServices services) {
                Constant.clServices = services;
            }
            @Override
            public void serviceDisconnected() {
            }
        });
    }

    npciPayBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            try {
                random=getSalt();
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            }

            payInfoArray = "[{\"name\":\"note\",\"value\":\"Verify UPI Pin
\"},{\"name\":\"account\",\"value\":\"XXXXXX8406\"},{\"name\":\"refId\",\"value\":\"U
PI932D731A084E4323B793F984D98068B5\"}]";
            salt
="{\"appId\":\"com.sbi.upi.test\",\"credType\":[\"setMpin\"],\"deviceId\":\"356152103
695358\",\"mobileNumber\":\"918169357679\",\"txnId\":[\"UPIEE18860197A4481
786F2DE9961F9644E\"],\"random\":\""+random+"\"}\"";
            String trustloc = null;
            try {
                trustloc = CreateTrust(salt);
            } catch (Exception e) {

            }

            trust = "{\"setMpin\":\""+trustloc+"\"}";


openNpciLib("{\"CredAllowed\":[{\"type\":\"PIN\",\"subtype\":\"MPIN\",\"dType\":\"N
UM\",\"dLength\":6},{\"type\":\"OTP\",\"subtype\":\"SMS\",\"dType\":\"NUM\",\"dLen
gth\":6}]}", salt, payInfoArray);
        }
    });


    final Button npciRegister = (Button) findViewById(R.id.register_getchallenge);

    npciRegister.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            String dataValue = Constant.clServices.getChallenge("initial",
```

```
UNQ_DEVICE_ID);

                String random= null;
                try {
                    random = getSalt();
                } catch (NoSuchAlgorithmException e) {
                    e.printStackTrace();
                }
                Log.d("TOken Recieved:", TOKEN);
                String hash = populateHMAC(getPackageName(), MOBILENO, TOKEN,
UNQ_DEVICE_ID,random);
                boolean regapp = Constant.clServices.registerApp(getPackageName(),
MOBILENO, UNQ_DEVICE_ID, hash,random);
                if(regapp){
                    Toast.makeText(MainActivity.this, "App is regstered",
Toast.LENGTH_LONG).show();

                }else{
                    Toast.makeText(MainActivity.this, "App is NOT regstered",
Toast.LENGTH_LONG).show();
                }
            }
        });
    }

     public byte[] hexStringToByteArray(String s) {
        byte[] b = new byte[s.length() / 2];

        for(int i = 0; i < b.length; ++i) {
            int index = i * 2;
            int v = Integer.parseInt(s.substring(index, index + 2), 16);
            b[i] = (byte)v;
        }

        return b;
    }

    public String populateHMAC(String app_id, String mobile, String token,
                              String deviceId,String random) {
        String hmac = null;
        try {
            CryptoUtils cryptLib = new CryptoUtils();
            String message = app_id + "|" + mobile + "|" + deviceId;
            Log.e(TAG, "PSP Hmac Msg - " +message);
            byte[] hmacBytes = cryptLib.aesEncrypt(
                    cryptLib.sha256Bytes(message,random),
                    hexStringToByteArray(token),random);
            hmac = Base64.encodeToString(hmacBytes, Base64.DEFAULT);
        } catch (Exception e) {
```

```
            Log.e(TAG, "populateHMAC ", e);
        }
        return hmac;
    }

    private static String getSalt() throws NoSuchAlgorithmException {
        SecureRandom random = new SecureRandom();
        byte[] salt = new byte[16];
        random.nextBytes(salt);
        return Base64.encodeToString(salt,Base64.NO_WRAP);
    }

    public void openNpciLib(final String cred, String salt, String payInfoArray) {
        CLRemoteResultReceiver remoteResultReceiver = new
CLRemoteResultReceiver(new ResultReceiver(new Handler()) {
            @Override
            protected void onReceiveResult(int resultCode, Bundle resultData) {
                super.onReceiveResult(resultCode, resultData);
                try{
                    Log.i("REsult",resultData.toString());
                    Toast.makeText(getApplicationContext(),"resultCode: "+resultCode+"
data: "+resultData.toString(),Toast.LENGTH_SHORT).show();
                    if(resultCode == 2) {

                        JSONObject data=new JSONObject();
                        data.put("status","0");
                        JSONObject atmRedirectJson = new JSONObject();
                        atmRedirectJson.put("data",data);
                        Constant.clServices.execute(atmRedirectJson.toString());
                    }
                    Constant.clServices.unbindService();
                }
                catch (Exception e){
                    //Do Nothing
                }
            }

        });


        JSONObject conf = new JSONObject();
      try {
            conf.put("payerBankName","Bank Example Bank");
            conf.put("color","#CD1C5F");
            conf.put("backgroundColor","#FFFFFF");
            conf.put("verifiedMerchant",verifiedMerchant());
            conf.put("forgotUpiPINEnabled",forgotUpiPinEnabled());
            conf.put("captureCardDetails","false");
```

```
            conf.put("bankImage", encodedImage);
            if(!TextUtils.isEmpty(otpNumAttempts.getText())){
                conf.put("numAttempts",otpNumAttempts.getText());
            }
            if(!TextUtils.isEmpty(imageUrl.getText())){
                conf.put("bankImageUrl",imageUrl.getText());
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }

        String langPref = languagePref;
        String configuration = conf.toString();

Constant.clServices.getCredential(keyCode,xml,cred,configuration,salt,payInfoArray
,trust,langPref,remoteResultReceiver);

    }

    private String CreateTrust(String salt) throws Exception, JSONException {
        String truststring = null;
        JSONObject jsonObject = new JSONObject(salt);
        String  txnIdArray = jsonObject.getJSONArray("txnId");
        String  txnId = txnIdArray.optString(0);
        String credTypearray  = jsonObject.getJSONArray("credType");
        String   credType = credTypearray.optString(0);
        String txnAmount =
jsonObject.optString(CLConstants.SALT_FIELD_TXN_AMOUNT);
        String appId = jsonObject.optString(CLConstants.SALT_FIELD_APP_ID);
        String deviceId = jsonObject.optString(CLConstants.SALT_FIELD_DEVICE_ID);
        String mobileNumber =
jsonObject.optString(CLConstants.SALT_FIELD_MOBILE_NUMBER);
        String payerAddr =
jsonObject.optString(CLConstants.SALT_FIELD_PAYER_ADDR);
        String payeeAddr =
jsonObject.optString(CLConstants.SALT_FIELD_PAYEE_ADDR);
        String random= jsonObject.optString(CLConstants.SALT_FIELD_RANDOM);

        try {StringBuilder trustParamBuilder = new StringBuilder(150);

            if(credType != null && !credType.isEmpty())

trustParamBuilder.append(credType).append(CLConstants.SALT_DELIMETER);

            if (txnId != null && !txnId.isEmpty())
                trustParamBuilder.append(txnId).append(CLConstants.SALT_DELIMETER);

            if (appId != null && !appId.isEmpty())
```

```
trustParamBuilder.append(appId).append(CLConstants.SALT_DELIMETER);

            if (mobileNumber != null && !mobileNumber.isEmpty())

trustParamBuilder.append(mobileNumber).append(CLConstants.SALT_DELIMETER
);

            if (deviceId != null && !deviceId.isEmpty())

trustParamBuilder.append(deviceId).append(CLConstants.SALT_DELIMETER);

            if (payerAddr != null && !payerAddr.isEmpty())

trustParamBuilder.append(payerAddr).append(CLConstants.SALT_DELIMETER);

            if (payeeAddr != null && !payeeAddr.isEmpty())

trustParamBuilder.append(payeeAddr).append(CLConstants.SALT_DELIMETER);

            if (txnAmount != null && !txnAmount.isEmpty())

trustParamBuilder.append(txnAmount).append(CLConstants.SALT_DELIMETER);

            // trim '|' if trust ends with '|'
            int i = trustParamBuilder.lastIndexOf(CLConstants.SALT_DELIMETER);
            if (i != -1
                    && i == trustParamBuilder.length() - 1) {
                trustParamBuilder.deleteCharAt(i);
            }

            String reconMsg = null;
            CryptoUtils cryptLib = new CryptoUtils();

            try {

                        byte[] trustBytes = cryptLib.aesEncrypt(
                        cryptLib.sha256Bytes(trustParamBuilder.toString(),random),
                        hexStringToByteArray(TOKEN),random);
                reconMsg = Base64.encodeToString(trustBytes, Base64.DEFAULT);
            } catch (Exception e) {
                throw e;
            }
            return reconMsg;

        } catch (Exception e) { }
        return truststring;
    }

   }
```

21.2 CryptLib.java

```
import android.util.Base64;
import android.util.Log;
import java.security.MessageDigest;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class CryptoUtils {
    private final static String LOG_TAG = CryptoUtils.class.getName();

    public static byte[] sha256Bytes(String s,String random) {
        try{
            byte[] randomBytes= Base64.decode(random,Base64.NO_WRAP);
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            digest.update(randomBytes);
            return digest.digest(s.getBytes("UTF-8"));
        } catch(Exception ex){
            Log.d(LOG_TAG, "Error generating hash ", ex);
        }
        return null;
    }

    public static byte[] aesEncrypt(final byte[] array, final byte[] array2,String
random) throws Exception {
        final SecretKeySpec secretKeySpec = new SecretKeySpec(array2, "AES");
        final IvParameterSpec ivParameterSpec = new
IvParameterSpec(Base64.decode(random,Base64.NO_WRAP));
        final Cipher instance = Cipher.getInstance("AES/CBC/PKCS5Padding");
        instance.init(Cipher.ENCRYPT_MODE, secretKeySpec, ivParameterSpec);
        return instance.doFinal(array);
    }


}
```