A Novel NFC based Secure Protocol for Merchant Transactions

SHAIK SHAKEEL AHAMAD1, AND MAJED ALOWAIDI2

1College of Computer and Information Sciences, Majmaah University, Al Majmaah, Kingdom of Saudi Arabia (e-mail [s.ahamad@mu.edu.sa](mailto:%20s.ahamad@mu.edu.sa))

2College of Computer and Information Sciences, Majmaah University, Al Majmaah, Kingdom of Saudi Arabia (e-mail: [m.alowaidi@mu.edu.sa](mailto:m.alowaidi@mu.edu.sa))

Corresponding author: Shaik Shakeel Ahamad (e-mail: [s.ahamad@mu.edu.sa](mailto:s.ahamad@mu.edu.sa)).

**Overview**

We Implemented ECDH Key exchange and AES encryption for proximity payments using Kotlin language in Android Studio (Kotlin is designed to interoperate fully with Java). We developed two apps one for Merchant and one for Customer. Customer App (MPA) and Merchant App (PPA) communicate through NFC. Shared symmetric key is agreed between the customer MPA and Bank server using ECDH Key exchange. Digital signatures are generated using ECDSA (digest algorithm used is SHA-256) and messages are encrypted using AES encryption algorithm. We implemented by use of Kotlin language in Android Studio. Kotlin is designed to interoperate fully with Java. We created an EC key pair (NIST P-256 aka secp256r1) at customer-bank and merchant-bank by using ECDH, we created a shared AES secret key. AES with GCM (Galois/Counter Mode) mode used for encryption and decryption of Customer Payment Data at MPA and PPA. Table 1 shows different environmental parameters.

Table 1. Implementation Environment and Parameters. Our implementation details and code can be found here:

|  |  |
| --- | --- |
| **Environment** | **Parameters** |
| Customer Mobile | NFC Enabled Android Mobile |
| Snapdragon 632 |
| 3GB RAM |
| Android v9.0 (minimum Android v6.0) |
| Merchant’s POS | NFC Enabled Android Mobile |
| Snapdragon 632 |
| 3GB RAM |
| Android v9.0 (minimum Android v6.0) |
| Bank server | Linux CentOS 7.8.2003 |
| Intel i7 9700k |
| 4GB RAM |
| 80GB SSD |
| Nginx Server 1.18.0 |
| PHP 7.2.31 |
| MariaDB 5.5.65 |
| Java (OpenJDK 1.8.0\_252) |

ECDH Key exchange between Customer (Mobile) and Bank Server to Generate AES Key.

Figure 1. shows the process of ECDH key exchange between the customer and the server. Both Customer and Server creates EC Key pair (Private key and Public Key). Server and Customer share Public keys and then by using Elliptic Curve Diffie-Hellman (ECDH) they create shared secret AES key.

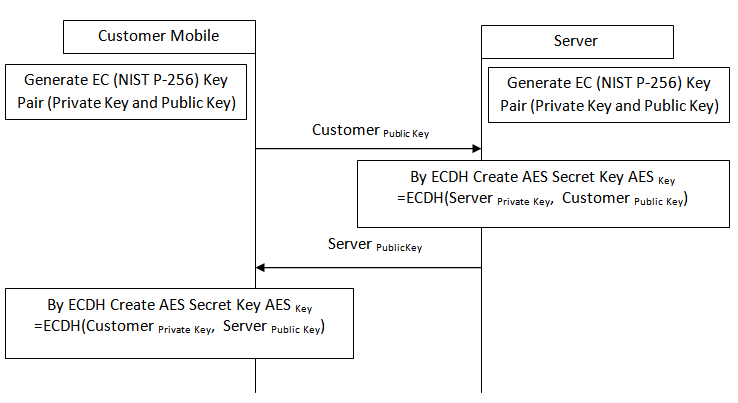


Figure 1. ECDH Key Exchange between Customer mobile and Bank Server.

ECDH Key exchange between Merchant (Mobile) and Bank Server to Generate AES Key.

Figure 2. shows the process of ECDH key exchange between the merchant and the server. Both Merchant and Server creates EC Key pair (Private key and Public Key). Server and Merchant share Public keys and then by using Elliptic Curve Diffie-Hellman (ECDH) they create shared secret AES key.

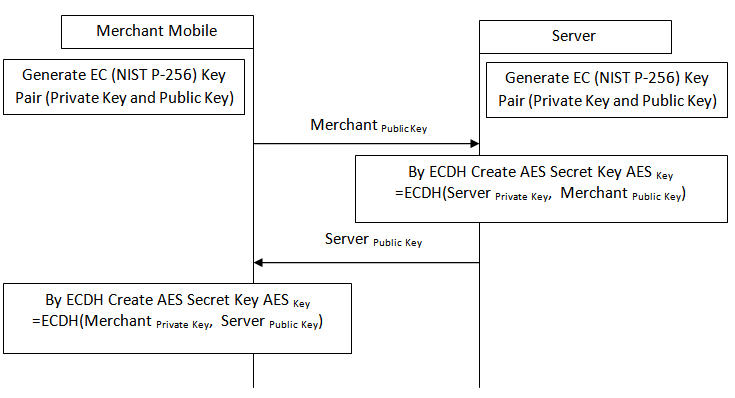


Figure 2. ECDH Key Exchange between Merchant mobile and Bank Server.

Customer Mobile, Merchant (Mobile) and Bank Server AES Encryption & Decryption.

Customer app encrypts payment data with shared AES and sends this encrypted payment data and customer ID to Merchant App. Merchant App encrypt payment data, customer ID and customer encrypted data with its own share AES key and sends encrypted data to Bank server. Every encryption or decryption involve initialization vector (IV) and this IV will be transferred between customer and server and merchant and server. Figure 3 shows encryption and decryption process.

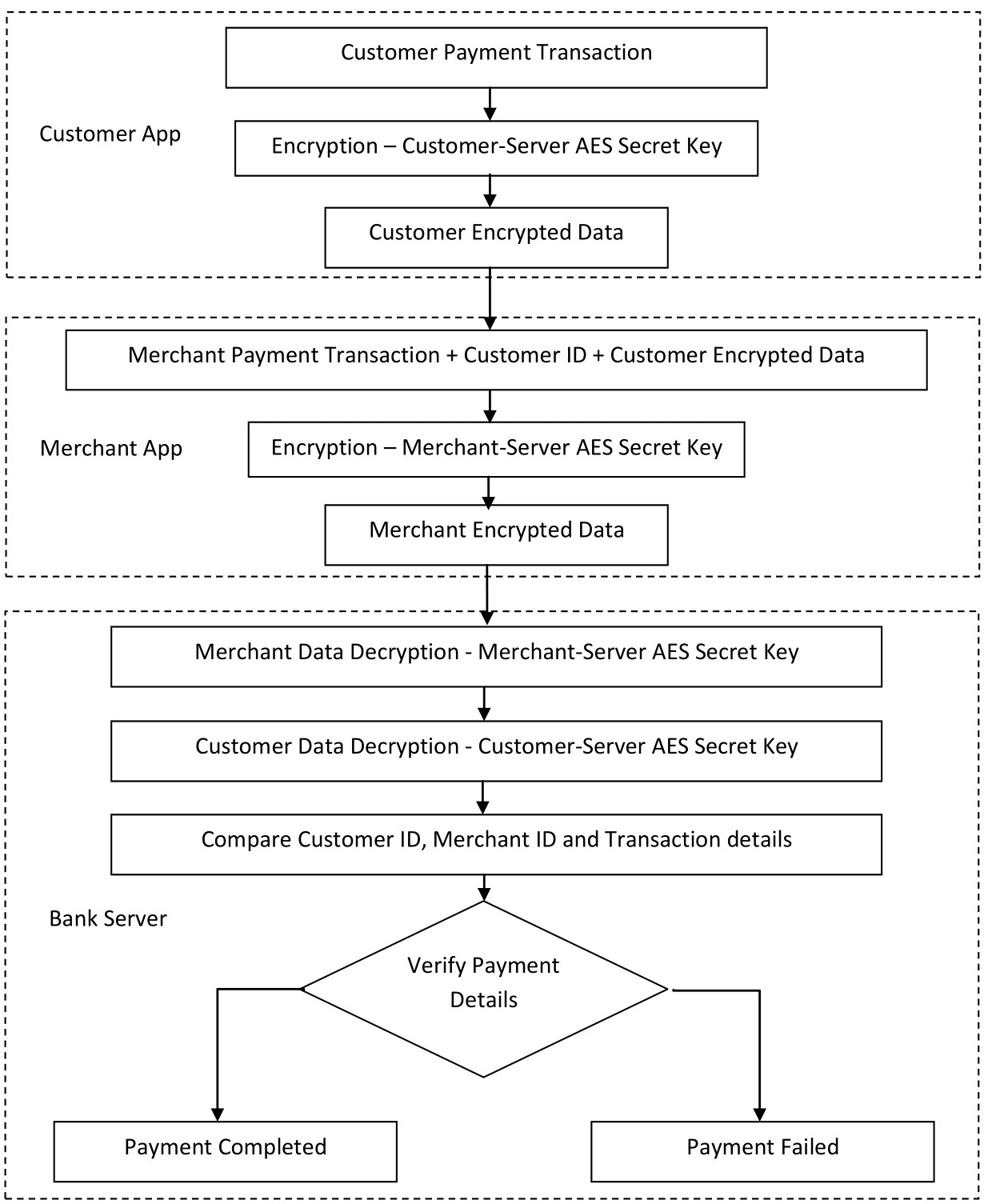


Figure 3. Encryption and Decryption of Process of Customer Payment Data

|  |  |
| --- | --- |
| Figure 4 shows ECDH key exchange process between the customer’s MPA and the bank server. Elliptic Curve (EC) private key and public key are generated at the Customer’s MPA and received at the bank’s public key. By using customer’s private key and Bank Server’s public key customer’s MPA generates shared AES key. Similarly bank server also generates shared AES key by using customer’s public key and bank server’s private key. | |
| Screenshot_20200803-033822.png | Screenshot_20200803-033827.png |
| Figure 4: ECDH key exchange between customer’s MPA and Bank server  Figure 5 shows the screen which contain customer information like customer name, customer ID and customer address. This screen contain button to begin NFC payment. | |
| Screenshot_20200803-033822.png  Figure 5. Customer information  Figure 6 shows ECDH key exchange process between merchant app and bank server. EC private key and public key generated at the merchant app and received the server public key. By using merchant private key and server public key merchant app generates shared AES key. Similarly server also generates shared AES key by using merchant public key and server private key. | |
|  | |
|  | |  |
| Screenshot_20200803-033822.png **Screenshot_20200803-042622.png**  Figure 6: ECDH key exchange between Merchant Payment Application and Bank server  Screenshot_20200803-033822.png  Figure 7. Merchant information  **Main Protocol/Transaction**  Figure 8 shows user authentication screen shot. To complete the payment customer need to authenticate either through Pin or Fingerprint. Only after successful authentication, app sends encrypted data to merchant app.  Screenshot_20200803-033822.png  Figure 8: Customerauthentication  Figure 9 shows the screen which gets merchant ID, merchant name and total bill amount from merchant app. And this screen contains button to pay bill, which sends encrypted data to merchant app.  Screenshot_20200803-033822.png  Figure 9. Customer app showing total bill amount and merchant details | |  |
| Figure 10 shows the screen which contain merchant transaction, customer ID and customer encrypted payment data. After successful payment completion at bank server this screen receives successful message from bank server and shows success status to merchant and sends successful message to customer app.  Screenshot_20200803-033822.png Screenshot_20200803-042735.png  Figure 10.PPA showing merchant transaction and payment status | |  |
| Figure 11 shows the screen which contain customer payment transaction and encrypted transaction. After successful payment completion at merchant this screen receives successful message from merchant app and shows success status to customer.  Screenshot_20200803-033822.png  Figure 11. MPA showing plain and encrypted transaction and payment status | |  |

**Customer Payment App.**

Customer Payment App involve a lot of classes. And only the code of ECDHCustomerServer class provided in this report.

**Code of ECDHCustomerServer Class**

**package** com.webmah.customerpaymentapp  
  
**import** android.app.Activity  
**import** android.app.KeyguardManager  
**import** android.content.Context  
**import** android.content.Intent  
**import** android.content.SharedPreferences  
**import** android.net.ConnectivityManager  
**import** android.os.Bundle  
**import** android.util.Base64  
**import** android.util.Log  
**import** android.view.View  
**import** android.widget.Toast  
**import** androidx.appcompat.app.AppCompatActivity  
**import** androidx.lifecycle.*lifecycleScope***import** kotlinx.android.synthetic.main.activity\_ecdhcustomerserver.\*  
**import** kotlinx.coroutines.Dispatchers  
**import** kotlinx.coroutines.launch  
**import** kotlinx.coroutines.withContext  
**import** org.json.JSONException  
**import** org.json.JSONObject  
**import** java.io.\*  
**import** java.net.HttpURLConnection  
**import** java.net.URL  
**import** java.security.\*  
**import** java.security.spec.ECGenParameterSpec  
**import** java.security.spec.PKCS8EncodedKeySpec  
**import** java.security.spec.X509EncodedKeySpec  
**import** javax.crypto.Cipher  
**import** javax.crypto.KeyAgreement  
**import** javax.crypto.SecretKey  
**import** javax.crypto.spec.GCMParameterSpec  
**import** javax.crypto.spec.SecretKeySpec  
**import** javax.net.ssl.HttpsURLConnection  
  
  
**class** ECDHCustomerServer : AppCompatActivity() {  
  
 **private lateinit var keyguardManager**: KeyguardManager  
 **private lateinit var keyPair**: KeyPair  
 **private lateinit var keyAES**: SecretKey  
 **private lateinit var signatureResult**: String  
 **private lateinit var enMessage**: String  
 **private val TAG** = **"MCCP"  
 private val iv** = **"123456789abcdefh"**.*toByteArray*()  
 **private lateinit var clientPrivateKey**: PrivateKey  
 **private lateinit var clientPublicKey**: PublicKey  
 **private lateinit var serverPublicKey**: PublicKey  
 **private lateinit var clientAES**: SecretKey  
 **private lateinit var serverResponse**: String  
  
  
 **override fun** onCreate(savedInstanceState: Bundle?) {  
 **super**.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_ecdhcustomerserver*)  
  
 **keyguardManager** = getSystemService(Context.*KEYGUARD\_SERVICE*) **as** KeyguardManager  
  
 *//Check if lock screen has been set up. Just displaying a Toast here but it shouldn't allow the user to go forward.* **if** (!**keyguardManager**.*isDeviceSecure*) {  
 Toast.makeText(**this**, **"Secure lock screen hasn't set up."**, Toast.*LENGTH\_LONG*).show()  
 }  
  
 **val** TRANSACTION = *intent*.getStringExtra(**"TRANSACTION"**)  
  
 *//showAuthenticationScreen()  
 //Check if the EC public private keys already exists to avoid creating them again* checkNetworkConnection()  
  
 **val** sharedPreference: SharedPreferences = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
 editor.clear()  
 editor.commit()  
  
 **if** (!checkECKeysExists()) {  
 generateECKeys()  
 }  
  
 **if** (!checkServerPubKeyExists()) {  
 getServerECPublicKey()  
 }  
  
 accessApp.setOnClickListener **{  
 val** intent = Intent(**this**, CustomerInfo::**class**.*java*)  
 startActivity(intent)  
 **}** }  
  
 **private fun** checkECKeysExists(): Boolean {  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"clientPrivateKey"**) && sharedPreference.contains(**"clientPublicKey"**)){  
 *// decode the base64 encoded string* **val** check = sharedPreference.getString(**"clientPublicKey"**, **"no"**)  
 **if**(check == **"no"**)  
 {  
 **return false** }  
  
 *// decode the base64 encoded string* **val** pukey: ByteArray = Base64.decode(sharedPreference.getString(**"clientPublicKey"**, **"no"**), Base64.*DEFAULT*)  
 **val** keySpec = X509EncodedKeySpec(pukey)  
 **val** keyFactory = KeyFactory.getInstance(**"EC"**)  
 **clientPublicKey** = keyFactory.generatePublic(keySpec)  
  
 **val** prkey: ByteArray = Base64.decode(sharedPreference.getString(**"clientPrivateKey"**, **"no"**), Base64.*DEFAULT*)  
 **val** keySpec1 = PKCS8EncodedKeySpec(prkey)  
 **val** keyFactory1 = KeyFactory.getInstance(**"EC"**)  
 **clientPrivateKey** = keyFactory1.generatePrivate(keySpec1)  
  
 **return true** }  
 **return false** }  
  
 **private fun** generateECKeys() {  
  
 **val** keyGen = KeyPairGenerator.getInstance(**"EC"**)  
 keyGen.initialize(ECGenParameterSpec(**"secp256r1"**), SecureRandom())  
 **val** pair = keyGen.generateKeyPair()  
 **clientPrivateKey** = pair.*private* **clientPublicKey** = pair.*public* **val** prkey = Base64.encodeToString(**clientPrivateKey**.*encoded*, Base64.*DEFAULT*)  
 **val** pkey = Base64.encodeToString(**clientPublicKey**.*encoded*, Base64.*DEFAULT*)  
 System.*out*.println(pkey)  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
  
 client\_pri.*text* = **"Customer EC Private Key: $**prkey**"** client\_pub.*text* = **"Customer EC Public Key: $**pkey**"** editor.putString(**"clientPrivateKey"**,prkey)  
 editor.putString(**"clientPublicKey"**,pkey)  
 editor.commit()  
 }  
  
 **private fun** checkServerPubKeyExists(): Boolean {  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"serverPublicKey"**)){  
 *// decode the base64 encoded string* **val** check = sharedPreference.getString(**"serverPublicKey"**, **"no"**)  
 **if**(check == **"no"**)  
 {  
 **return false** }  
  
 *// decode the base64 encoded string* **val** pukey: ByteArray = Base64.decode(sharedPreference.getString(**"serverPublicKey"**, **"no"**), Base64.*DEFAULT*)  
 **val** keySpec = X509EncodedKeySpec(pukey)  
 **val** keyFactory = KeyFactory.getInstance(**"EC"**)  
 **serverPublicKey** = keyFactory.generatePublic(keySpec)  
  
 **return true** }  
 **return false** }  
  
 @Throws(JSONException::**class**)  
 **private fun** getServerECPublicKey() {  
 *// clear text result* **serverResponse** = **"no"  
  
 if** (checkNetworkConnection()) {  
 **val** ckey = Base64.encodeToString(**clientPublicKey**.*encoded*, Base64.*DEFAULT*)  
 **val** jsonObject = JSONObject()  
 jsonObject.accumulate(**"clientPublicKeyEC"**, ckey)  
 *lifecycleScope*.*launch* **{  
 val** result = httpPost(**"https://webmah.com/customerpaymentapp/ServerAuthAtClient.php"**, jsonObject)  
  
 **val** responseparts = **serverResponse**.*split*(**"-------"**)  
 **val** serpukey: ByteArray = Base64.decode(responseparts[0], Base64.*DEFAULT*)  
 **val** keySpec = X509EncodedKeySpec(serpukey)  
 **val** keyFactory = KeyFactory.getInstance(**"EC"**)  
 **serverPublicKey** = keyFactory.generatePublic(keySpec)  
  
 **val** spkey = Base64.encodeToString(**serverPublicKey**.*encoded*, Base64.*DEFAULT*)  
 System.*out*.println(**"ServerPublic Key:"**+spkey)  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
  
 server\_pub.*text* = **"Bank Server EC Public Key: $**spkey**"** editor.putString(**"serverPublicKey"**,spkey)  
 editor.commit()  
  
 **if** (!checkSharedAESKeyExists()) {  
 getSharedAESKey()  
 }  
  
 *//decrypt signature* **val** cipher = Cipher.getInstance(*TRANSFORMATION*)  
 *//We decode the signature value  
 //serverensign\_txt.text = "Server Encrypted Signature: $responseparts[1]"* **val** ensign: ByteArray = Base64.decode(responseparts[1], Base64.*DEFAULT*)  
 **val** serverIV: ByteArray = Base64.decode(responseparts[2], Base64.*DEFAULT*)  
  
 cipher.init(Cipher.*DECRYPT\_MODE*, **clientAES**, GCMParameterSpec(128, serverIV))  
 **val** decodedData: ByteArray = cipher.doFinal(ensign)  
  
 **var** ssignstr = Base64.encodeToString(decodedData, Base64.*DEFAULT*)  
 *//System.out.println("Server Signature:"+ssignstr)* verifyServerSign(decodedData)  
 **}** }  
 **else** Toast.makeText(**this**, **"Not Connected!"**, Toast.*LENGTH\_SHORT*).show()  
  
 }  
  
 **private fun** checkSharedAESKeyExists(): Boolean {  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"clientAES"**)){  
 *// decode the base64 encoded string* **var** seck = sharedPreference.getString(**"clientAES"**, **"no"**)  
 **if**(seck == **"no"**)  
 {  
 **return false** }  
  
 **val** secKey: ByteArray = Base64.decode(seck, Base64.*DEFAULT*)  
 **clientAES** = SecretKeySpec(secKey, 0, secKey.**size**, **"AES"**)  
  
 **return true** }  
 **return false** }  
  
 **private fun** getSharedAESKey() {  
 **val** secretKeyAES: SecretKey? = generateSharedSecret(**clientPrivateKey**, **serverPublicKey**)  
  
 **if** (secretKeyAES != **null**) {  
 **clientAES** = secretKeyAES  
 }  
  
 **val** aeskey = Base64.encodeToString(secretKeyAES?.*encoded*, Base64.*DEFAULT*)  
 System.*out*.println(**"Client AESKey:"**+aeskey)  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*,Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
  
 client\_aes.*text* = **"ECDH Generated AES Key:$**aeskey**"** editor.putString(**"clientAES"**,aeskey)  
 editor.commit()  
 }  
  
 **private fun** generateSharedSecret(privateKey: PrivateKey?, publicKey: PublicKey?): SecretKey? {  
 **return try** {  
 **val** keyAgreement: KeyAgreement = KeyAgreement.getInstance(**"ECDH"**)  
 keyAgreement.init(privateKey)  
 keyAgreement.doPhase(publicKey, **true**)  
 **val** key: ByteArray = keyAgreement.generateSecret()  
 *//String ke = Base64.getEncoder().encodeToString(key);  
 //System.out.println(ke);* SecretKeySpec(key, 0, key.**size**, **"AES"**)  
 } **catch** (e: java.lang.Exception) {  
 e.printStackTrace()  
 **null** }  
 }  
  
 **private fun** verifyServerSign(serversign: ByteArray?) {  
 **try** {  
  
 *//val signature: ByteArray = Base64.decode(serversign, Base64.DEFAULT)* **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"clientPublicKey"**)) {  
 *// decode the base64 encoded string* **var** seck = sharedPreference.getString(**"clientPublicKey"**, **"no"**)  
 System.*out*.println(**"Client PublicKey:"**+seck);  
  
 *//We check if the signature is valid. We use ECDSA algorithm along SHA-256 digest algorithm* **val** isValid: Boolean = Signature.getInstance(**"SHA256withECDSA"**).*run* **{** initVerify(**serverPublicKey**)  
 **if** (seck != **null**) {  
 update(Base64.decode(seck, Base64.*DEFAULT*))  
 }  
 verify(serversign)  
 **}  
 if** (isValid) {  
 System.*out*.println(**"valid: Server Authenticated "**);  
 *//server\_verified.text = "Server Authentication Successful at Client"* verifyClintAtServer()  
  
 } **else** {  
 System.*out*.println(**"notvalid: Server Authentication Failed"**);  
 *//server\_verified.text = "Server Authentication Failed at Client"* }  
  
 }  
  
 } **catch** (e : Exception){  
 **throw** RuntimeException(e)  
 }  
 }  
  
 @Throws(JSONException::**class**)  
 **private fun** verifyClintAtServer() {  
 *// clear text result* **serverResponse** = **"no"  
  
 val** cipher = Cipher.getInstance(*TRANSFORMATION*)  
  
 **val** parameterSpec = GCMParameterSpec(128, **iv**)  
 cipher.init(Cipher.*ENCRYPT\_MODE*, **clientAES**, parameterSpec)  
  
 **val** bytes = cipher.doFinal(createClientSign())  
 **val** clientENSign = Base64.encodeToString(bytes, Base64.*DEFAULT*)  
  
 *//client\_sign.text = "Client Signature: $clientENSign"* **val** ivs = Base64.encodeToString(**iv**, Base64.*DEFAULT*)  
  
  
 **if** (checkNetworkConnection()) {  
 **val** jsonObject = JSONObject()  
 jsonObject.accumulate(**"iv"**, ivs)  
 jsonObject.accumulate(**"clientENSign"**, clientENSign)  
 *lifecycleScope*.*launch* **{  
 val** result = httpPost(**"https://webmah.com/customerpaymentapp/ClientAuthAtServer.php"**, jsonObject)  
 *//client\_verified.text = serverResponse* **if**(**serverResponse**.*contains*(**"Successfully"**, ignoreCase = **true**))  
 {  
 accessApp.*visibility* = View.*VISIBLE*; *//To set visible* }  
 **}** }  
 **else** Toast.makeText(**this**, **"Not Connected!"**, Toast.*LENGTH\_SHORT*).show()  
  
 }  
  
 **private fun** createClientSign(): ByteArray? {  
 **try** {  
 *//val signature: ByteArray = Base64.decode(serversign, Base64.DEFAULT)* **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"serverPublicKey"**)) {  
 *// decode the base64 encoded string* **var** seck = sharedPreference.getString(**"serverPublicKey"**, **"no"**)  
  
 *//We sign the data with the private key. We use ECDAS algorithm along SHA-256 digest algorithm* **val** signature: ByteArray? = Signature.getInstance(**"SHA256withECDSA"**).*run* **{** initSign(**clientPrivateKey**)  
 update(Base64.decode(seck, Base64.*DEFAULT*))  
 sign()  
 **}  
 return** signature  
 }  
 } **catch** (e : Exception){  
 **throw** RuntimeException(e)  
 }  
 **return null** }  
  
  
 **private fun** showAuthenticationScreen() {  
 *//This will open a screen to enter the user credentials (fingerprint, pin, pattern). We can display a custom title and description* **val** intent: Intent? = **keyguardManager**.createConfirmDeviceCredentialIntent(**"User Authentication"**,  
 **"To be able to use this Smart Meter Secure App we need to confirm your identity. Please enter your pin/pattern or scan your fingerprint"**)  
 **if** (intent != **null**) {  
 startActivityForResult(intent, *REQUEST\_CODE\_FOR\_CREDENTIALS*)  
 }  
 }  
  
 **override fun** onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
 **if** (requestCode == *REQUEST\_CODE\_FOR\_CREDENTIALS*) {  
 **if** (resultCode == Activity.*RESULT\_OK*) {  
 } **else** {  
 Toast.makeText(**this**, **"Authentication failed."**, Toast.*LENGTH\_SHORT*).show()  
 }  
 }  
 }  
  
  
  
 @Throws(IOException::**class**, JSONException::**class**)  
 **private suspend fun** httpPost(myUrl: String, jsonObject: JSONObject): String {  
  
 **val** result = withContext(Dispatchers.**IO**) **{  
 val** url = URL(myUrl)  
 *// 1. create HttpURLConnection* **val** conn = url.openConnection() **as** HttpsURLConnection  
 conn.*requestMethod* = **"POST"** conn.setRequestProperty(**"Content-Type"**, **"application/json; charset=utf-8"**)  
  
 *// 2. build JSON object  
 //val jsonObject = buidJsonObject()  
  
 // 3. add JSON content to POST request body* setPostRequestContent(conn, jsonObject)  
  
 *// 4. make POST request to the given URL* conn.connect()  
  
 *// 5. return response message* conn.*responseMessage* + **""  
  
 if** (conn.*responseCode* == HttpsURLConnection.*HTTP\_OK*) {  
 **val** stream = BufferedInputStream(conn.*inputStream*)  
 **serverResponse** = readStream(inputStream = stream)  
 } **else** {  
 **serverResponse** = **"Problem in Getting Server Response"** }  
  
 **}  
 return** result.toString()  
 }  
  
 **private fun** checkNetworkConnection(): Boolean {  
 **val** connMgr = getSystemService(Context.*CONNECTIVITY\_SERVICE*) **as** ConnectivityManager  
  
 **val** networkInfo = connMgr.*activeNetworkInfo* **val** isConnected: Boolean = **if**(networkInfo != **null**) networkInfo.isConnected() **else false  
 if** (networkInfo != **null** && isConnected) {  
 *// show "Connected" & type of network "WIFI or MOBILE"* howIsConnected.*text* = **"Connected "** + networkInfo.*typeName* } **else** {  
 *// show "Not Connected"* howIsConnected.*text* = **"Not Connected"** }  
 **return** isConnected  
 }  
  
 @Throws(JSONException::**class**)  
 **private fun** buidJsonObject(): JSONObject {  
  
 **val** pkey = Base64.encodeToString(**clientPublicKey**.*encoded*, Base64.*DEFAULT*)  
 **val** skey = Base64.encodeToString(**clientAES**.*encoded*, Base64.*DEFAULT*)  
 **val** ivs = Base64.encodeToString(**iv**, Base64.*DEFAULT*)  
  
  
 **val** jsonObject = JSONObject()  
 jsonObject.accumulate(**"aeskey"**, skey)  
 jsonObject.accumulate(**"iv"**, ivs)  
 jsonObject.accumulate(**"encryptedTransaction"**, **enMessage**)  
 jsonObject.accumulate(**"publickey"**, pkey)  
 jsonObject.accumulate(**"signature"**, **signatureResult**)  
  
 **return** jsonObject  
 }  
  
 @Throws(IOException::**class**)  
 **private fun** setPostRequestContent(conn: HttpURLConnection, jsonObject: JSONObject) {  
  
 **val** os = conn.*outputStream* **val** writer = BufferedWriter(OutputStreamWriter(os, **"UTF-8"**))  
 writer.write(jsonObject.toString())  
 Log.i(**TAG**, jsonObject.toString())  
 writer.flush()  
 writer.close()  
 os.close()  
 }  
  
 **private fun** readStream(inputStream: BufferedInputStream): String {  
 **val** bufferedReader = BufferedReader(InputStreamReader(inputStream))  
 **val** stringBuilder = StringBuilder()  
 bufferedReader.*forEachLine* **{** stringBuilder.append(**it**) **}  
 return** stringBuilder.toString()  
 }  
  
}  
  
**private const val** *REQUEST\_CODE\_FOR\_CREDENTIALS* = 1  
**private const val** *TRANSFORMATION* = **"AES/GCM/NoPadding"  
private const val** *SHAREDLOCATION* = **"CUSTOMERPAYMENTAPP"**

**Merchant Payment Process App.**

Merchant Payment Process App involve a lot of classes. And only the code of ECDHMerchantServer class provided in this report.

**Code of ECDHMerchantServer Class**

**package** com.webmah.merchantpaymentprocessapp  
  
**import** android.app.Activity  
**import** android.app.KeyguardManager  
**import** android.content.Context  
**import** android.content.Intent  
**import** android.content.SharedPreferences  
**import** android.net.ConnectivityManager  
**import** android.os.Bundle  
**import** android.util.Base64  
**import** android.util.Log  
**import** android.view.View  
**import** android.widget.Toast  
**import** androidx.appcompat.app.AppCompatActivity  
**import** androidx.lifecycle.*lifecycleScope***import** kotlinx.android.synthetic.main.activity\_ecdhmerchantserver.\*  
**import** kotlinx.coroutines.Dispatchers  
**import** kotlinx.coroutines.launch  
**import** kotlinx.coroutines.withContext  
**import** org.json.JSONException  
**import** org.json.JSONObject  
**import** java.io.\*  
**import** java.net.HttpURLConnection  
**import** java.net.URL  
**import** java.security.\*  
**import** java.security.spec.ECGenParameterSpec  
**import** java.security.spec.PKCS8EncodedKeySpec  
**import** java.security.spec.X509EncodedKeySpec  
**import** javax.crypto.Cipher  
**import** javax.crypto.KeyAgreement  
**import** javax.crypto.SecretKey  
**import** javax.crypto.spec.GCMParameterSpec  
**import** javax.crypto.spec.SecretKeySpec  
**import** javax.net.ssl.HttpsURLConnection  
  
  
**class** ECDHMerchantServer : AppCompatActivity() {  
  
 **private lateinit var keyguardManager**: KeyguardManager  
 **private lateinit var keyPair**: KeyPair  
 **private lateinit var keyAES**: SecretKey  
 **private lateinit var signatureResult**: String  
 **private lateinit var enMessage**: String  
 **private val TAG** = **"MCMP"  
 private val iv** = **"123456789abcdefh"**.*toByteArray*()  
 **private lateinit var clientPrivateKey**: PrivateKey  
 **private lateinit var clientPublicKey**: PublicKey  
 **private lateinit var serverPublicKey**: PublicKey  
 **private lateinit var clientAES**: SecretKey  
 **private lateinit var serverResponse**: String  
  
  
 **override fun** onCreate(savedInstanceState: Bundle?) {  
 **super**.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_ecdhmerchantserver*)  
  
 **keyguardManager** = getSystemService(Context.*KEYGUARD\_SERVICE*) **as** KeyguardManager  
  
 *//Check if lock screen has been set up. Just displaying a Toast here but it shouldn't allow the user to go forward.* **if** (!**keyguardManager**.*isDeviceSecure*) {  
 Toast.makeText(**this**, **"Secure lock screen hasn't set up."**, Toast.*LENGTH\_LONG*).show()  
 }  
  
 **val** TRANSACTION = *intent*.getStringExtra(**"TRANSACTION"**)  
  
 *//showAuthenticationScreen()  
 //Check if the EC public private keys already exists to avoid creating them again* checkNetworkConnection()  
  
 **val** sharedPreference: SharedPreferences = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
 editor.clear()  
 editor.commit()  
  
 **if** (!checkECKeysExists()) {  
 generateECKeys()  
 }  
  
 **if** (!checkServerPubKeyExists()) {  
 getServerECPublicKey()  
 }  
  
 accessApp.setOnClickListener **{  
 val** intent = Intent(**this**, MerchantInfo::**class**.*java*)  
 startActivity(intent)  
 **}** }  
  
 **private fun** checkECKeysExists(): Boolean {  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"clientPrivateKey"**) && sharedPreference.contains(**"clientPublicKey"**)){  
 *// decode the base64 encoded string* **val** check = sharedPreference.getString(**"clientPublicKey"**, **"no"**)  
 **if**(check == **"no"**)  
 {  
 **return false** }  
  
 *// decode the base64 encoded string* **val** pukey: ByteArray = Base64.decode(sharedPreference.getString(**"clientPublicKey"**, **"no"**), Base64.*DEFAULT*)  
 **val** keySpec = X509EncodedKeySpec(pukey)  
 **val** keyFactory = KeyFactory.getInstance(**"EC"**)  
 **clientPublicKey** = keyFactory.generatePublic(keySpec)  
  
 **val** prkey: ByteArray = Base64.decode(sharedPreference.getString(**"clientPrivateKey"**, **"no"**), Base64.*DEFAULT*)  
 **val** keySpec1 = PKCS8EncodedKeySpec(prkey)  
 **val** keyFactory1 = KeyFactory.getInstance(**"EC"**)  
 **clientPrivateKey** = keyFactory1.generatePrivate(keySpec1)  
  
 **return true** }  
 **return false** }  
  
 **private fun** generateECKeys() {  
  
 **val** keyGen = KeyPairGenerator.getInstance(**"EC"**)  
 keyGen.initialize(ECGenParameterSpec(**"secp256r1"**), SecureRandom())  
 **val** pair = keyGen.generateKeyPair()  
 **clientPrivateKey** = pair.*private* **clientPublicKey** = pair.*public* **val** prkey = Base64.encodeToString(**clientPrivateKey**.*encoded*, Base64.*DEFAULT*)  
 **val** pkey = Base64.encodeToString(**clientPublicKey**.*encoded*, Base64.*DEFAULT*)  
 System.*out*.println(pkey)  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
  
 client\_pri.*text* = **"Merchant EC Private Key: $**prkey**"** client\_pub.*text* = **"Merchant EC Public Key: $**pkey**"** editor.putString(**"clientPrivateKey"**,prkey)  
 editor.putString(**"clientPublicKey"**,pkey)  
 editor.commit()  
 }  
  
 **private fun** checkServerPubKeyExists(): Boolean {  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"serverPublicKey"**)){  
 *// decode the base64 encoded string* **val** check = sharedPreference.getString(**"serverPublicKey"**, **"no"**)  
 **if**(check == **"no"**)  
 {  
 **return false** }  
  
 *// decode the base64 encoded string* **val** pukey: ByteArray = Base64.decode(sharedPreference.getString(**"serverPublicKey"**, **"no"**), Base64.*DEFAULT*)  
 **val** keySpec = X509EncodedKeySpec(pukey)  
 **val** keyFactory = KeyFactory.getInstance(**"EC"**)  
 **serverPublicKey** = keyFactory.generatePublic(keySpec)  
  
 **return true** }  
 **return false** }  
  
 @Throws(JSONException::**class**)  
 **private fun** getServerECPublicKey() {  
 *// clear text result* **serverResponse** = **"no"  
  
 if** (checkNetworkConnection()) {  
 **val** ckey = Base64.encodeToString(**clientPublicKey**.*encoded*, Base64.*DEFAULT*)  
 **val** jsonObject = JSONObject()  
 jsonObject.accumulate(**"clientPublicKeyEC"**, ckey)  
 *lifecycleScope*.*launch* **{  
 val** result = httpPost(**"https://webmah.com/customerpaymentapp/ServerAuthAtClient.php"**, jsonObject)  
  
 **val** responseparts = **serverResponse**.*split*(**"-------"**)  
 **val** serpukey: ByteArray = Base64.decode(responseparts[0], Base64.*DEFAULT*)  
 **val** keySpec = X509EncodedKeySpec(serpukey)  
 **val** keyFactory = KeyFactory.getInstance(**"EC"**)  
 **serverPublicKey** = keyFactory.generatePublic(keySpec)  
  
 **val** spkey = Base64.encodeToString(**serverPublicKey**.*encoded*, Base64.*DEFAULT*)  
 System.*out*.println(**"ServerPublic Key:"**+spkey)  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
  
 server\_pub.*text* = **"Bank Server EC Public Key: $**spkey**"** editor.putString(**"serverPublicKey"**,spkey)  
 editor.commit()  
  
 **if** (!checkSharedAESKeyExists()) {  
 getSharedAESKey()  
 }  
  
 *//decrypt signature* **val** cipher = Cipher.getInstance(*TRANSFORMATION*)  
 *//We decode the signature value  
 //serverensign\_txt.text = "Server Encrypted Signature: $responseparts[1]"* **val** ensign: ByteArray = Base64.decode(responseparts[1], Base64.*DEFAULT*)  
 **val** serverIV: ByteArray = Base64.decode(responseparts[2], Base64.*DEFAULT*)  
  
 cipher.init(Cipher.*DECRYPT\_MODE*, **clientAES**, GCMParameterSpec(128, serverIV))  
 **val** decodedData: ByteArray = cipher.doFinal(ensign)  
  
 **var** ssignstr = Base64.encodeToString(decodedData, Base64.*DEFAULT*)  
 *//System.out.println("Server Signature:"+ssignstr)* verifyServerSign(decodedData)  
 **}** }  
 **else** Toast.makeText(**this**, **"Not Connected!"**, Toast.*LENGTH\_SHORT*).show()  
  
 }  
  
 **private fun** checkSharedAESKeyExists(): Boolean {  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"clientAES"**)){  
 *// decode the base64 encoded string* **var** seck = sharedPreference.getString(**"clientAES"**, **"no"**)  
 **if**(seck == **"no"**)  
 {  
 **return false** }  
  
 **val** secKey: ByteArray = Base64.decode(seck, Base64.*DEFAULT*)  
 **clientAES** = SecretKeySpec(secKey, 0, secKey.**size**, **"AES"**)  
  
 **return true** }  
 **return false** }  
  
 **private fun** getSharedAESKey() {  
 **val** secretKeyAES: SecretKey? = generateSharedSecret(**clientPrivateKey**, **serverPublicKey**)  
  
 **if** (secretKeyAES != **null**) {  
 **clientAES** = secretKeyAES  
 }  
  
 **val** aeskey = Base64.encodeToString(secretKeyAES?.*encoded*, Base64.*DEFAULT*)  
 System.*out*.println(**"Client AESKey:"**+aeskey)  
 **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*,Context.*MODE\_PRIVATE*)  
 **var** editor = sharedPreference.edit()  
  
 client\_aes.*text* = **"ECDH Generated AES Key:$**aeskey**"** editor.putString(**"clientAES"**,aeskey)  
 editor.commit()  
 }  
  
 **private fun** generateSharedSecret(privateKey: PrivateKey?, publicKey: PublicKey?): SecretKey? {  
 **return try** {  
 **val** keyAgreement: KeyAgreement = KeyAgreement.getInstance(**"ECDH"**)  
 keyAgreement.init(privateKey)  
 keyAgreement.doPhase(publicKey, **true**)  
 **val** key: ByteArray = keyAgreement.generateSecret()  
 *//String ke = Base64.getEncoder().encodeToString(key);  
 //System.out.println(ke);* SecretKeySpec(key, 0, key.**size**, **"AES"**)  
 } **catch** (e: java.lang.Exception) {  
 e.printStackTrace()  
 **null** }  
 }  
  
 **private fun** verifyServerSign(serversign: ByteArray?) {  
 **try** {  
  
 *//val signature: ByteArray = Base64.decode(serversign, Base64.DEFAULT)* **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"clientPublicKey"**)) {  
 *// decode the base64 encoded string* **var** seck = sharedPreference.getString(**"clientPublicKey"**, **"no"**)  
 System.*out*.println(**"Client PublicKey:"**+seck);  
  
 *//We check if the signature is valid. We use ECDSA algorithm along SHA-256 digest algorithm* **val** isValid: Boolean = Signature.getInstance(**"SHA256withECDSA"**).*run* **{** initVerify(**serverPublicKey**)  
 **if** (seck != **null**) {  
 update(Base64.decode(seck, Base64.*DEFAULT*))  
 }  
 verify(serversign)  
 **}  
 if** (isValid) {  
 System.*out*.println(**"valid: Server Authenticated "**);  
 *//server\_verified.text = "Server Authentication Successful at Client"* verifyClintAtServer()  
  
 } **else** {  
 System.*out*.println(**"notvalid: Server Authentication Failed"**);  
 *//server\_verified.text = "Server Authentication Failed at Client"* }  
  
 }  
  
 } **catch** (e : Exception){  
 **throw** RuntimeException(e)  
 }  
 }  
  
 @Throws(JSONException::**class**)  
 **private fun** verifyClintAtServer() {  
 *// clear text result* **serverResponse** = **"no"  
  
 val** cipher = Cipher.getInstance(*TRANSFORMATION*)  
  
 **val** parameterSpec = GCMParameterSpec(128, **iv**)  
 cipher.init(Cipher.*ENCRYPT\_MODE*, **clientAES**, parameterSpec)  
  
 **val** bytes = cipher.doFinal(createClientSign())  
 **val** clientENSign = Base64.encodeToString(bytes, Base64.*DEFAULT*)  
  
 *//client\_sign.text = "Client Signature: $clientENSign"* **val** ivs = Base64.encodeToString(**iv**, Base64.*DEFAULT*)  
  
  
 **if** (checkNetworkConnection()) {  
 **val** jsonObject = JSONObject()  
 jsonObject.accumulate(**"iv"**, ivs)  
 jsonObject.accumulate(**"clientENSign"**, clientENSign)  
 *lifecycleScope*.*launch* **{  
 val** result = httpPost(**"https://webmah.com/customerpaymentapp/ClientAuthAtServer.php"**, jsonObject)  
 *//client\_verified.text = serverResponse* **if**(**serverResponse**.*contains*(**"Successfully"**, ignoreCase = **true**))  
 {  
 accessApp.*visibility* = View.*VISIBLE*; *//To set visible* }  
 **}** }  
 **else** Toast.makeText(**this**, **"Not Connected!"**, Toast.*LENGTH\_SHORT*).show()  
  
 }  
  
 **private fun** createClientSign(): ByteArray? {  
 **try** {  
 *//val signature: ByteArray = Base64.decode(serversign, Base64.DEFAULT)* **val** sharedPreference = getSharedPreferences(*SHAREDLOCATION*, Context.*MODE\_PRIVATE*)  
 *// && sharedPreference.contains("serverPublicKey")* **if**(sharedPreference.contains(**"serverPublicKey"**)) {  
 *// decode the base64 encoded string* **var** seck = sharedPreference.getString(**"serverPublicKey"**, **"no"**)  
  
 *//We sign the data with the private key. We use ECDAS algorithm along SHA-256 digest algorithm* **val** signature: ByteArray? = Signature.getInstance(**"SHA256withECDSA"**).*run* **{** initSign(**clientPrivateKey**)  
 update(Base64.decode(seck, Base64.*DEFAULT*))  
 sign()  
 **}  
 return** signature  
 }  
 } **catch** (e : Exception){  
 **throw** RuntimeException(e)  
 }  
 **return null** }  
  
  
 **private fun** showAuthenticationScreen() {  
 *//This will open a screen to enter the user credentials (fingerprint, pin, pattern). We can display a custom title and description* **val** intent: Intent? = **keyguardManager**.createConfirmDeviceCredentialIntent(**"User Authentication"**,  
 **"To be able to use this Smart Meter Secure App we need to confirm your identity. Please enter your pin/pattern or scan your fingerprint"**)  
 **if** (intent != **null**) {  
 startActivityForResult(intent, *REQUEST\_CODE\_FOR\_CREDENTIALS*)  
 }  
 }  
  
 **override fun** onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
 **if** (requestCode == *REQUEST\_CODE\_FOR\_CREDENTIALS*) {  
 **if** (resultCode == Activity.*RESULT\_OK*) {  
 } **else** {  
 Toast.makeText(**this**, **"Authentication failed."**, Toast.*LENGTH\_SHORT*).show()  
 }  
 }  
 }  
  
  
  
 @Throws(IOException::**class**, JSONException::**class**)  
 **private suspend fun** httpPost(myUrl: String, jsonObject: JSONObject): String {  
  
 **val** result = withContext(Dispatchers.**IO**) **{  
 val** url = URL(myUrl)  
 *// 1. create HttpURLConnection* **val** conn = url.openConnection() **as** HttpsURLConnection  
 conn.*requestMethod* = **"POST"** conn.setRequestProperty(**"Content-Type"**, **"application/json; charset=utf-8"**)  
  
 *// 2. build JSON object  
 //val jsonObject = buidJsonObject()  
  
 // 3. add JSON content to POST request body* setPostRequestContent(conn, jsonObject)  
  
 *// 4. make POST request to the given URL* conn.connect()  
  
 *// 5. return response message* conn.*responseMessage* + **""  
  
 if** (conn.*responseCode* == HttpsURLConnection.*HTTP\_OK*) {  
 **val** stream = BufferedInputStream(conn.*inputStream*)  
 **serverResponse** = readStream(inputStream = stream)  
 } **else** {  
 **serverResponse** = **"Problem in Getting Server Response"** }  
  
 **}  
 return** result.toString()  
 }  
  
 **private fun** checkNetworkConnection(): Boolean {  
 **val** connMgr = getSystemService(Context.*CONNECTIVITY\_SERVICE*) **as** ConnectivityManager  
  
 **val** networkInfo = connMgr.*activeNetworkInfo* **val** isConnected: Boolean = **if**(networkInfo != **null**) networkInfo.isConnected() **else false  
 if** (networkInfo != **null** && isConnected) {  
 *// show "Connected" & type of network "WIFI or MOBILE"* howIsConnected.*text* = **"Connected "** + networkInfo.*typeName* } **else** {  
 *// show "Not Connected"* howIsConnected.*text* = **"Not Connected"** }  
 **return** isConnected  
 }  
  
 @Throws(JSONException::**class**)  
 **private fun** buidJsonObject(): JSONObject {  
  
 **val** pkey = Base64.encodeToString(**clientPublicKey**.*encoded*, Base64.*DEFAULT*)  
 **val** skey = Base64.encodeToString(**clientAES**.*encoded*, Base64.*DEFAULT*)  
 **val** ivs = Base64.encodeToString(**iv**, Base64.*DEFAULT*)  
  
  
 **val** jsonObject = JSONObject()  
 jsonObject.accumulate(**"aeskey"**, skey)  
 jsonObject.accumulate(**"iv"**, ivs)  
 jsonObject.accumulate(**"encryptedTransaction"**, **enMessage**)  
 jsonObject.accumulate(**"publickey"**, pkey)  
 jsonObject.accumulate(**"signature"**, **signatureResult**)  
  
 **return** jsonObject  
 }  
  
 @Throws(IOException::**class**)  
 **private fun** setPostRequestContent(conn: HttpURLConnection, jsonObject: JSONObject) {  
  
 **val** os = conn.*outputStream* **val** writer = BufferedWriter(OutputStreamWriter(os, **"UTF-8"**))  
 writer.write(jsonObject.toString())  
 Log.i(**TAG**, jsonObject.toString())  
 writer.flush()  
 writer.close()  
 os.close()  
 }  
  
 **private fun** readStream(inputStream: BufferedInputStream): String {  
 **val** bufferedReader = BufferedReader(InputStreamReader(inputStream))  
 **val** stringBuilder = StringBuilder()  
 bufferedReader.*forEachLine* **{** stringBuilder.append(**it**) **}  
 return** stringBuilder.toString()  
 }  
  
}  
  
**private const val** *REQUEST\_CODE\_FOR\_CREDENTIALS* = 1  
**private const val** *TRANSFORMATION* = **"AES/GCM/NoPadding"  
private const val** *SHAREDLOCATION* = **"MERCHANTPAYMENTAPP"**