

Mobile Devices Security

Fernando García Sanz(100346043)

Francisco Manuel Colmenar Lamas (100346094)

April 22, 2019



Contents

1	Introduction	4
2	Encryption of the CredHub app database	5
3	HTTPS and Certificate Pinning	11
3.1	HTTPS	11
3.2	Analyze the connection with Wireshark	14
3.3	Certificate Pinning	19
4	Conclusion	23

List of Figures

1	Pin Secure	5
2	<i>KeyStore</i> Creation	6
3	Alias Input	6
4	Keys Generation	7
5	Password Variables	7
6	Password Generator	8
7	Database Access	8
8	Password Request	9
9	Password Log	9
10	Query Outcome	10
11	HTTPS URL	11
12	TrustManager	12
13	SSLContext	13
14	HTTPS Authentication Header	13
15	HTTPS Authentication Variables	13
16	Running <i>Credhub</i>	14
17	Packets obtained with Wireshark	15
18	Client Handshake protocol packet	15
19	Server Handshake protocol packet	16
20	Handshake Protocol: Server Hello	16
21	Handshake Protocol: Certificate	17

22	Subject public key info	17
23	Subject public key info	18
24	Handshake protocol algorithm	18
25	Handshake Protocol: Server Key Exchange	18
26	Handshake Protocol: Server Key Exchange	19
27	android:networkSecurityConfig added in the Manifest	19
28	xml folder	19
29	raw folder	20
30	Trust anchors	20
31	Pin set and trust anchors	21
32	ConnectServer modified for certificate pinning	21
33	TLS packets using certificate pinning	22
34	Rogue certificate	22
35	Wireshark unknown certificate alert	22

1 Introduction

It is known that encrypting the information inside the devices is key in order to ensure its privacy. Besides, the popularity of this practise is growing among companies which are focus on ensuring a correct level of data security.

In this third and last module some techniques to encrypt data, especially sensitive data, will be described in order to gain a deeper understanding about the importance of this practise in a real scenario.

In order to achieve this goal several libraries and core concepts will be needed to be mastered. Some of the main concepts of this module are Android KeyStore provider, symmetric encryption keys an SQLCipher.

Furthermore, the last section of this module will consist on describing the importance of implementing in the correct way the HTTPS protocol. Once that its importance has been explained, an implementation of this protocol for CredHub will be accomplished and described in this report.

2 Encryption of the CredHub app database

The first task to be developed in this practice consists of the encryption of the database previously implemented in the application in past releases.

The database encryption does not mean that the app has to lose any of its functionalities, but only enhancing the security of its contents.

Before performing this activity, the database structure has been modified to properly accomplish the *DAO* (Data Access Object) structure, allowing an easier and modular implementation of the encryption.

To perform the database encryption the process has been split into five different phases:

1. Pin code for unlocking the device:

The first improvement implemented in this application release is the protection by pin code check.

By means of the class *KeyguardManager* it can be checked if the device implements any kind of protection by pin code, pattern..., and decide whether continue or not with the application execution according to that. The pin protection is checked as follows:

```
public boolean isSecure(){
    KeyguardManager keyguardManager = (KeyguardManager) getSystemService(Activity.KEYGUARD_SERVICE);
    boolean secure = false; // Default value false for secured device

    try {
        secure = keyguardManager.isDeviceSecure(); // Check if device is secure
    }
    catch (Exception e){
        e.printStackTrace();
    }

    return secure;
}
```

Figure 1: Pin Secure

This function returns a boolean which says whether the device implements or not pin protection. If it is implemented, the application will continue with its normal execution, if not, it will exit and go to home screen.

2. Android *KeyStore* storage creation:

Android *KeyStore* is an Android feature which eases keys secure storage. It restricts the access to the keys only to the application that created them and its secured by means of the device pin.

Android *KeyStore* creation can be seen here:

```
public static final String ANDROID_KEYSTORE = "AndroidKeyStore";

public void loadKeyStore() {
    try {
        keyStore = getInstance(ANDROID_KEYSTORE);
        keyStore.load( param: null);
    } catch (Exception e) {
        // TODO: Handle this appropriately in your app
        e.printStackTrace();
    }
}
```

Figure 2: *KeyStore* Creation

3. Public and private keys generation by means of an alias:

Now that the *KeyStore* is up and running, it is time to create the pair of keys to be stored in it. To do so, first an alias is needed to identify those keys.

This alias is introduced by the application user, and it will only be requested once, the first time the application is started, since it will be later stored into Android *Shared Preferences*, and accessed from there anytime it is needed.

Thus, once the user is logged in for first time, a screen requesting for the alias introduction is displayed.

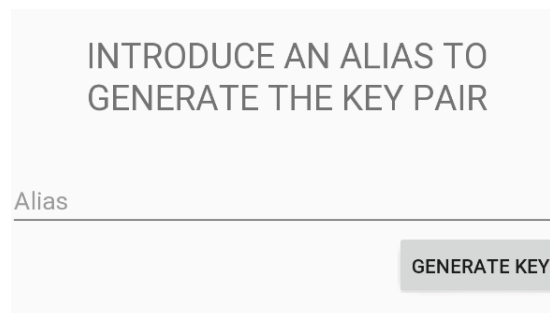
The image shows a mobile application interface with a light gray background. At the top, the text "INTRODUCE AN ALIAS TO GENERATE THE KEY PAIR" is displayed in a bold, dark gray font. Below this text is a text input field with the placeholder label "Alias" in a smaller, gray font. To the right of the input field is a rectangular button with the text "GENERATE KEY" in a bold, dark gray font.

Figure 3: Alias Input

Once the user has introduced the alias and pressed the button, both public and private keys are generated linked to that alias and stored into the *KeyStore*, as it can be seen in the image below.

```

public void generateNewKeyPair(String alias, Context context) throws Exception {

    Calendar start = Calendar.getInstance();
    Calendar end = Calendar.getInstance();
    // expires 1 year from today
    end.add(Calendar.YEAR, 1);

    KeyPairGeneratorSpec spec = new
        KeyPairGeneratorSpec.Builder(context)
            .setAlias(alias)
            .setSubject(new X500Principal( name: "CN=" + alias))
            .setSerialNumber(BigInteger.TEN)
            .setStartDate(start.getTime())
            .setEndDate(end.getTime())
            .build();

    // use the Android keystore
    KeyPairGenerator gen =
        KeyPairGenerator.getInstance( algorithm: "RSA", ANDROID_KEYSTORE);
    gen.initialize(spec);
    // generates the keypair
    gen.generateKeyPair();
}

```

Figure 4: Keys Generation

4. Random password generation, encryption and storage:

Once the keys have been generated, it is time to use them to encrypt the database which will store the credentials.

The database is generated once the user performs a successful login into the application, but it cannot be encrypted until an alias is provided for the key generation.

The database encryption is possible due to the implementation of *SQL Cipher* instead of just *SQLite*. This implementation incorporates all the functions and features of *SQLite* but adds them an extra of security encrypting the database by means of a password.

In this case, to enhance the security, the provided password is random, making that each installation of the application will provide a different 64 characters password randomly generated. The random password is alphanumeric and is generated, again, once the user performs the first successful login. The password generator function can be seen below:

```

static final String DATA = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
static Random RANDOM = new Random();

```

Figure 5: Password Variables


```

public static String randomString(int len) {
    StringBuilder sb = new StringBuilder(len);

    for (int i = 0; i < len; i++) {
        sb.append(DATA.charAt(RANDOM.nextInt(DATA.length())));
    }
    return sb.toString();
}

```

Figure 6: Password Generator

The generated password is then stored in the *Shared Preferences*. Later on, once the keys have been generated by means of the input alias, this password is encrypted using the public key, as *RSA* algorithm is used. The encrypted password will then substitute the non-encrypted one into *Shared Preferences*.

Once this operations have been performed, when the first operation with the database is executed, the password is decrypted and used as the password for the database access.

5. Transparent encryption/decryption once alias is given:

From now on, anytime the database has to be accessed, the password has to be provided. Therefore, the process to follow is recovering the encrypted password from *Shared Preferences* and decrypting it using the generated private key, obtaining then the password in plaintext, which will enable the access to the database.

This can be seen in the following picture:

```

try {
    database = UserDBHelper.getInstance(context).getReadableDatabase(
        keyStoreClass.decryptPassword(sprefs.getString(Alias, s1: ""), sprefs));
}
catch (Exception e) {
    e.printStackTrace();
}

```

Figure 7: Database Access

All these operations are transparent for the user, being the only difference the alias input during the first time the application is used. Password and keys are generated during this first use, securely stored and the database is created and encrypted by means of those password and public key and decrypted using the private one.

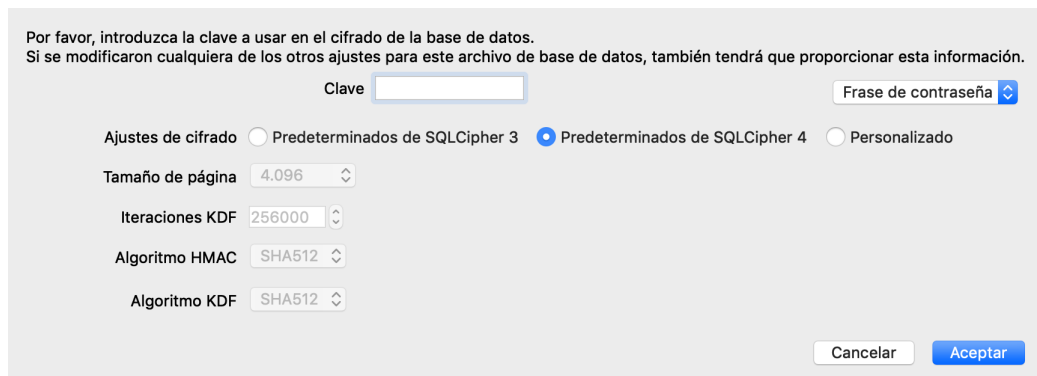
All this operations will run in the application background, so the user may think that in this release he is using the same application only with the alias change, but the application and

data security will have increased exponentially.

Now that the entire database has been secured, it can be tested to check if the database is properly protected or not.

To do so, the database file *Credential.db* has been extracted from the device. Previously, this archive could be opened using a program such as *DB Browser for SQLite* and freely explored. Nevertheless, if the same operation is performed now, the outcome will be different.

Now, a password is requested by the program in order to open the database file, and if it is not properly introduced, the file content is not possible of being read. This can be seen in the following picture:



Por favor, introduzca la clave a usar en el cifrado de la base de datos.
Si se modificaron cualquiera de los otros ajustes para este archivo de base de datos, también tendrá que proporcionar esta información.

Clave Frase de contraseña ▾

Ajustes de cifrado ☐ Predeterminados de SQLCipher 3 ☒ Predeterminados de SQLCipher 4 ☐ Personalizado

Tamaño de página 4.096 ▾

Iteraciones KDF 256000 ▾

Algoritmo HMAC SHA512 ▾

Algoritmo KDF SHA512 ▾

Cancelar Aceptar

Figure 8: Password Request

Remember that the password is a randomly generated 64 characters string which is after encrypted, so it is impossible accessing to its value from the final code. What has been done is modifying the code adding a log with the decrypted password for this specific case, which can be seen below:

```
2019-04-19 13:43:50.814 6624-6624/com.example.credhub I/PASSWORD: 1U6KKEU3E2L7RBH5B8USJRJW8GHAPLZXCGL0ECXS43TXRPPIDLGYM4A24MVL28Y2
```

Figure 9: Password Log

This password is then introduced using *SQL Cipher 3* cipher settings, accessing to the database file.

Finally, the following *SQL* query has been performed to extract the content of the database file, obtaining the final content, which can be seen in the following image:

[illegible]

Figure 10: Query Outcome

3 HTTPS and Certificate Pinning

In this section, *CredHub* is going to be modified according to its networking. To be more precise, *HTTPS* is going to be set up instead of HTTP. Once that this is done, using *Wireshark* the communications between the web service and the application are going to be analyzed in order to determine the difference with the previous assignment. Finally, a *Certificate Pinning* will be implemented so as to increase the level of trust in the identity of the remote web server.

3.1 HTTPS

To start with, HTTPS is going to be used in the connection between the server and the client of *Credhub*. Thanks to the implementation of HTTPS, when the HTTP request is done and sent to the server, it is sent using the Secure Socket Layer or SSL. The main objective of this system is avoiding or reducing the possibility of a Man in the Middle attack which consists on someone, the attacker, listening to the conversation between the client and the server.

Firstly, the command which is needed in order to run the server using HTTPS is different than the one used in previous assignments. In this case the following command is the one which is going to be used.

```
sudo java - -add - modules java.se.ee - jar SDM_WebRepo.jar https + auth (1)
```

Once this is done the server is running waiting only for HTTPS requests.

After that, the next thing to be done is changing the protocol from HTTP to HTTPS in the client side. In order to accomplish it first of all the URL needs to be changed into the following one.

```
static final String URL = "https://10.0.2.2/SDM/WebRepo?wsdl";
```

Figure 11: HTTPS URL

In our implementation all the networking operations are done in the java class called *"Talk-ToServer"*. To be more specific, the HTTP protocol is managed in the method *"connectServer"* which is the one in charge of setting the headers of the request with the needed authentication

parameters as well as setting the *androidHttpTransport*. Due to the fact that the HTTP protocol is to be switched to HTTPS, this method is the one which needs to be modified.

To start with, a *TrustManager* needs to be created. This *TrustManager* is not going to validate the certificate chains nor the client or the server. This is known to be a bad practise due to the fact that the application do not check whether the server is a trustful server or not. Consequently, this part will be modified in the last subsection of this section of the report. In this subsection a *trust-anchors* and *pin-set* will be implemented in order to fix the previously described security issues.

```
// Create a trust manager who trust all the certificates
TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        // Do not validate certificate chains
        @Override public java.security.cert.X509Certificate[] getAcceptedIssuers() {
            return new X509Certificate[0]; }
        // Do not check the client
        @Override public void checkClientTrusted(
            java.security.cert.X509Certificate[] certs, String authType) { }
        // Do not check the server
        @Override public void checkServerTrusted(
            java.security.cert.X509Certificate[] certs, String authType) {}
    }
}
```

Figure 12: TrustManager

Once that the *TrustManager* has been created, the next thing to be created is the *SSLContext*. This is one of the key features of HTTPS and what makes it secure. Furthermore, the *SSLContext* is initialized and a *Socket Factory* is set to the *HttpsURLConnection*. Besides, error checking are implemented with different messages in order to improve the debug stage.

```

HttpsURLConnection.setDefaultHostnameVerifier ((hostname, session) -> true);
// Initialize TLS context
SSLContext sc = null;
try {
    sc = SSLContext.getInstance("TLSv1.2"); // Set the SSL context
    sc.init( km: null, trustAllCerts, new java.security.SecureRandom());
} catch (KeyManagementException e) {
    Log.i( tag: "connectServer", msg: "ERROR - KeyManagementException - " + e.toString());
}
catch (NoSuchAlgorithmException e) {
    Log.i( tag: "connectServer", msg: "ERROR - NoSuchAlgorithmException - " + e.toString());
}
if (sc != null) {
    HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory()); // Set the socket factory
}

```

Figure 13: SSLContext

Finally, once that the *SSLContext* has been correctly initialized, the last step is setting the authentication variables in the header.

```

// Set HTTPS URL
androidHttpTransport = new HttpTransportSE(URL);
// Basic authentication
headerList_basicAuth = new ArrayList<>();
String strUserPass = Constant.BASIC_AUTH_USERNAME + ":" + Constant.BASIC_AUTH_PASSWORD; // Setting the String with the username and the password
headerList_basicAuth.add(new HeaderProperty("Authorization", "Basic " +
    org.kobjects.base64.Base64.encode(strUserPass.getBytes()))); // Set the Header with the Authentication information

```

Figure 14: HTTPS Authentication Header

Furthermore, "*Constant.BASIC_AUTH_USERNAME*" and "*Constant.BASIC_AUTH_PASSWORD*" are defined in the java class "*Constant*" where several constant variables are defined. This class is created in order to decrease the complexity in the main java classes by means of extracting those variables which values have to be defined previously to the execution of the application. These two variables have the following Strings assigned.

```

// Authentication values for server connection
static final String BASIC_AUTH_USERNAME = "sdm";
static final String BASIC_AUTH_PASSWORD = "repo4droid";

```

Figure 15: HTTPS Authentication Variables

3.2 Analyze the connection with Wireshark

Once that *Credhub* is able of running using HTTPS instead of HTTP, the communications between the application and the server are going to be analyzed. The objective of this process is to determine if the data is being correctly ciphered or not. Furthermore, in order to determine it the result obtained in this section will be compared with the result in the previous assignment where HTTP was the protocol used.

The first thing to be done is initializing the emulator and the server. The server needs to be initialized with the command described in the previous section (1). Besides, the emulator can be initialized through Android Studio or using the command line. In this case the first option is going to be the chosen one.

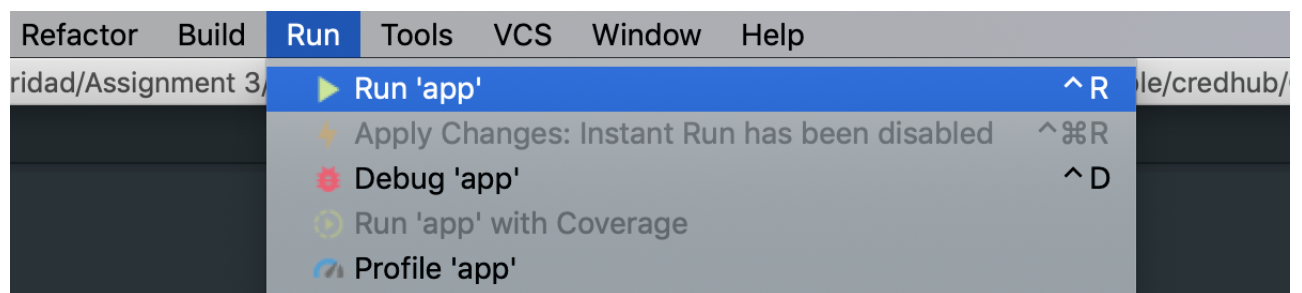


Figure 16: Running *Credhub*

When both the client and the server of *Credhub* are running, *Wireshark* has to be launch. Once that this is done everything is ready for listening to the packets sent from client to server and vice versa.

Once that *Wireshark* is launched in this example it is going to be listening in the interface *lo0*, also called *loopback*, due to the fact that the client and the server are running in the same computer.

Furthermore, in order to listen to the packets exchanged from the server to the client the last one has to perform any action in the Application which needs contacting with the server. In this case, the client is going to display the records stored in the server.

The packets obtained using *Wireshark* referring to this action are the following ones.

No.	Time	Source	Destination	Protocol	Length	Info
91	3.898620	127.0.0.1	127.0.0.1	TLSv1...	210	Client Hello
92	3.898679	127.0.0.1	127.0.0.1	TCP	56	443 → 50786 [ACK] Seq=1 Ack=155 Win=408128 Len=0 TSval=372867621 TSecr=372867621
93	3.907042	127.0.0.1	127.0.0.1	TLSv1...	3041	Server Hello, Certificate, Server Key Exchange, Server Hello Done
94	3.907065	127.0.0.1	127.0.0.1	TCP	56	50786 → 443 [ACK] Seq=155 Ack=2986 Win=405312 Len=0 TSval=372867629 TSecr=372867629
95	3.912560	127.0.0.1	127.0.0.1	TLSv1...	182	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
96	3.912613	127.0.0.1	127.0.0.1	TCP	56	443 → 50786 [ACK] Seq=2986 Ack=281 Win=408000 Len=0 TSval=372867634 TSecr=372867634
97	3.916207	127.0.0.1	127.0.0.1	TLSv1...	62	Change Cipher Spec
98	3.916237	127.0.0.1	127.0.0.1	TCP	56	50786 → 443 [ACK] Seq=281 Ack=2992 Win=405248 Len=0 TSval=372867637 TSecr=372867637
99	3.916472	127.0.0.1	127.0.0.1	TLSv1...	101	Encrypted Handshake Message
100	3.916489	127.0.0.1	127.0.0.1	TCP	56	50786 → 443 [ACK] Seq=281 Ack=3037 Win=405248 Len=0 TSval=372867637 TSecr=372867637
101	3.917546	127.0.0.1	127.0.0.1	TLSv1...	740	Application Data
102	3.917567	127.0.0.1	127.0.0.1	TCP	56	443 → 50786 [ACK] Seq=3037 Ack=965 Win=407296 Len=0 TSval=372867638 TSecr=372867638
103	3.921546	127.0.0.1	127.0.0.1	TLSv1...	208	Application Data
104	3.921569	127.0.0.1	127.0.0.1	TCP	56	50786 → 443 [ACK] Seq=965 Ack=3189 Win=405056 Len=0 TSval=372867641 TSecr=372867641
105	3.922258	127.0.0.1	127.0.0.1	TLSv1...	353	Application Data
106	3.922283	127.0.0.1	127.0.0.1	TCP	56	50786 → 443 [ACK] Seq=965 Ack=3486 Win=404800 Len=0 TSval=372867641 TSecr=372867641
107	3.922523	127.0.0.1	127.0.0.1	TLSv1...	90	Application Data

Figure 17: Packets obtained with Wireshark

As it can be seen, in this case instead of been displayed the post operation, like it happened in the previous assignment, the key negotiation is taking place. Firstly, the client contacts with the server with the packet number 91. This is the starting point of the *Handshake* protocol.

Transport Layer Security

- ▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
 - Content Type: Handshake (22)
 - Version: TLS 1.0 (0x0301)
 - Length: 149
- ▼ Handshake Protocol: Client Hello
 - Handshake Type: Client Hello (1)
 - Length: 145
 - Version: TLS 1.2 (0x0303)

Figure 18: Client Handshake protocol packet

After that the client has sent to the server the packet, it is time for the server to answer it. Moreover, inside the packet which the server sends there are four different elements of the *Handshake* protocol. They are: the *Server Hello*, the *Certificate* from the server, the *Server Key Exchange* and the *Server Hello Done*.

Transport Layer Security

- ▼ TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages
 - Content Type: Handshake (22)
 - Version: TLS 1.2 (0x0303)
 - Length: 2980
 - ▶ Handshake Protocol: Server Hello
 - ▶ Handshake Protocol: Certificate
 - ▶ Handshake Protocol: Server Key Exchange
 - ▶ Handshake Protocol: Server Hello Done

Figure 19: Server Handshake protocol packet

According to *Server Hello*, several parameters are sent with it such as the TLS version, the session ID and the Compression Method.

```
Handshake Protocol: Server Hello
Handshake Type: Server Hello (2)
Length: 81
Version: TLS 1.2 (0x0303)
▶ Random: 8eb249bb58470ef06be4a5e7115442d0020666544af200c6...
Session ID Length: 32
Session ID: 1dc40c7e6178da3d19a477133a58ca941ca70da35020d69c...
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
Compression Method: null (0)
Extensions Length: 9
▶ Extension: renegotiation_info (len=1)
▶ Extension: extended_master_secret (len=0)
```

Figure 20: Handshake Protocol: Server Hello

Regarding the *Certificate*, all the information regarding to the server certificate is displayed. Its length, the validation of the certificate, the issuer and its serial number among other parameters.

```

Handshake Protocol: Certificate
  Handshake Type: Certificate (11)
  Length: 2554
  Certificates Length: 2551
  ▼ Certificates (2551 bytes)
    Certificate Length: 1170
    ▼ Certificate: 3082048e30820276a00302010202040a0143c2300d06092a...
      ▼ signedCertificate
        version: v3 (2)
        serialNumber: 167855042
        ► signature (sha256WithRSAEncryption)
        ► issuer: rdnSequence (0)
        ► validity
        ► subject: rdnSequence (0)
        ► subjectPublicKeyInfo
        ► extensions: 3 items
        ► algorithmIdentifier (sha256WithRSAEncryption)
        Padding: 0
        encrypted: 530f066a1b594d503fede2193a5b6013db4bdf4ee216ad6a...
      Certificate Length: 1375
    ► Certificate: 3082055b30820343a00302010202042674cdec300d06092a...

```

Figure 21: Handshake Protocol: Certificate

There are three parameters which are specially interesting which are the issuer, the public key information and the algorithm identifier.

In the issuer section the different details of the server certificate are showed. Thanks to this, more information about the certificate can be known such as the organization name of this certificate.

```

issuer: rdnSequence (0)
  ▼ rdnSequence: 6 items (id-at-commonName=SDM-CA,id-at-organizationa
    ► RDNSquence item: 1 item (id-at-countryName=ES)
    ► RDNSquence item: 1 item (id-at-stateOrProvinceName=Madrid)
    ► RDNSquence item: 1 item (id-at-localityName=Madrid)
    ► RDNSquence item: 1 item (id-at-organizationName=UC3M)
    ► RDNSquence item: 1 item (id-at-organizationalUnitName=UC3M)
    ► RDNSquence item: 1 item (id-at-commonName=SDM-CA)

```

Figure 22: Subject public key info

The public key information displays information about the algorithm used in order to encrypt the key, which in this case *RSA* is the chosen algorithm. In addition to this, the modulus and the

exponent of the public key is also sent in this packet.

```
▼ subjectPublicKeyInfo
  ▼ algorithm (rsaEncryption)
    Algorithm Id: 1.2.840.113549.1.1.1 (rsaEncryption)
  ▼ subjectPublicKey: 3082010a0282010100a09b10cd3767fbf75bfd7c6887bcd...
    modulus: 0x00a09b10cd3767fbf75bfd7c6887bcdde740d99a8b62638a...
    publicExponent: 65537
```

Figure 23: Subject public key info

The last parameter of the *Certificate* that is worth to mention is the *algorithmIdentifier*. In this case SHA 256 with *RSA* encryption is used.

```
▼ algorithmIdentifier (sha256WithRSAEncryption)
  Algorithm Id: 1.2.840.113549.1.1.11 (sha256WithRSAEncryption)
  Padding: 0
  encrypted: 530f066a1b594d503fede2193a5b6013db4bdf4ee216ad6a...
```

Figure 24: Handshake protocol algorithm

According to *Server Key Exchange* the parameters for using the algorithm Diffie-Hellman as well as the Signature parameters are sent to the client in order to be able of establishing the connection.

```
Handshake Protocol: Server Key Exchange
  Handshake Type: Server Key Exchange (12)
  Length: 329
  ▼ EC Diffie-Hellman Server Params
    Curve Type: named_curve (0x03)
    Named Curve: secp256r1 (0x0017)
    Pubkey Length: 65
    Pubkey: 04c4a988a4c799684de0ff6a54e914c023fbf911a925a3a4...
  ▼ Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
    Signature Hash Algorithm Hash: SHA512 (6)
    Signature Hash Algorithm Signature: RSA (1)
    Signature Length: 256
    Signature: 6205f74e720ea48415ebe65e7806480a295520ac5685a749...
```

Figure 25: Handshake Protocol: Server Key Exchange

Finally, once that the *Handshake* protocol has finished the transmission of data starts. They are the packets named as *Application Data*. If they are looked more closely it can be seen that the data inside them is encrypted and there is no way of reading the plain text data as it happened in

the previous assignment when HTTP was been used.

Transport Layer Security

▼ TLSv1.2 Record Layer: Application Data Protocol: http-over-tls
Content Type: Application Data (23)
Version: TLS 1.2 (0x0303)
Length: 679
Encrypted Application Data: 0000000000000001a5e70ba5924ef43b22c4de236aadac7c...

Figure 26: Handshake Protocol: Server Key Exchange

3.3 Certificate Pinning

In this section *Certificate Pinning* functionalities will be implemented in order to enhance the trust on the identity of the remote web server.

In order to implement *certificate pinning* in *Credhub* it is needed to modified the manifest file of the application. It has to be added the *"android:networkSecurityConfig"* attribute.

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="CredHub"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:networkSecurityConfig="@xml/network_security_config"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
```

Figure 27: android:networkSecurityConfig added in the Manifest

As it can be seen in the previous image, to *"android:networkSecurityConfig"* it is assigned a value which references to a xml file called *"network_security_config"*. This file is created in a new folder called *"xml"* inside the *"res"* folder.

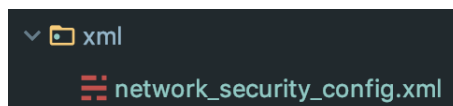


Figure 28: xml folder

Inside this new file the *certificate pinning* as well as the *trust-anchors* are going to be implement.

Firstly, the *trust-anchors* is going to be implemented. It defines the set of trust anchors, the given certificate in our case, for secure connections in the Application. Consequently, the trusted anchor need to be added locally to the project. The certificate "*sdm_web.cer*" is added into a new folder called "*raw*" which is located inside "*res*".

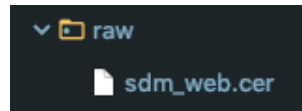


Figure 29: raw folder

Once that the certificate has been added to the project folders it is time to modified the xml file. The field *trust-anchors* is added indicating the path where the trusted certificate is located.

```
<domain-config>
  <domain includeSubdomains="true">10.0.2.2</domain>
  <trust-anchors>
    <certificates src="@raw/sdm_web"/>
  </trust-anchors>
</domain-config>
```

Figure 30: Trust anchors

Furthermore, a *pin-set* need to be defined. It has to contains the SHA-256 hashes of the public keys of the accepted certificates of the application, which in this case need to be the certificate of the server. Consequently, for a secure connection to be trusted once that a *pin-set* is defined, one of the public keys in the chain of trust must be the same as the one defined at *pin-set*.

It is recommendable to include a backup set of pins in case that the server rotates its public keys. However, as in this case is a simple example and it is not a real case no backup pin is set.

```

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="true">10.0.2.2</domain>
    <trust-anchors>
      <certificates src="@raw/sdm_web"/>
    </trust-anchors>
    <pin-set>
      <pin digest="SHA-256">ku9bDfzWHgsHF2uuzfEdv4dst2QxhzyYUcaZQDYJo80=</pin>
    </pin-set>
  </domain-config>
</network-security-config>

```

Figure 31: Pin set and trust anchors

According to the class *TalkToServer*, the method *connectServer* where the HTTPS connection is done needs to be modified. In the first subsection of this section a *trustManager* which accepts all the certificates no matter their origin was implemented. Now, once that the *pin set* and *trust anchors* are implemented in order to them to work correctly this method needs to be modified. The trust manager is going to be deleted and the method is going to be reduced to the following code.

```

private void connectServer() {
    // Set HTTPS URL
    androidHttpTransport = new HttpTransportSE(URL);
    // Basic authentication
    headerList_basicAuth = new ArrayList<>();
    String strUserPass = Constant.BASIC_AUTH_USERNAME + ":" + Constant.BASIC_AUTH_PASSWORD; // Setting the String with the username and the password
    headerList_basicAuth.add(new HeaderProperty("Authorization", "Basic " +
        org.kobjects.base64.Base64.encode(strUserPass.getBytes()))); // Set the Header with the Authentication information
}

```

Figure 32: ConnectServer modified for certificate pinning

In order to verify the effectiveness and the successful implementation of *certificate pinning* it is going to be challenged against a *rogue* server. Consequently, in order to be able to verify the correct implementation of *certificate pinning* the server needs to be launched using the following command.

```

sudo java --add-modules java.se.ee -jar SDM_WebRepo.jar https + auth + rogue (2)

```

Once that the server is running, the application is launched and it is tried to access to the records stored in the server. Once that this is done the application open the page of the read

records but as it does not establish the connection no records are showed.

If Wireshark is used in order to have a closer insight of what is happening the following packets are found.

85	3.099848	127.0.0.1	127.0.0.1	TLSv1...	3041	Server Hello, Certificate, Server Key Exchange, Server Hello Done
86	3.099896	127.0.0.1	127.0.0.1	TCP	56	49762 → 443 [ACK] Seq=155 Ack=2986 Win=405312 Len=0 TSval=348477958 TSecr=348477958
87	3.125235	127.0.0.1	127.0.0.1	TLSv1...	63	Alert (Level: Fatal, Description: Certificate Unknown)

Figure 33: TLS packets using certificate pinning

As it can be seen the connection is not established. If the first packet is opened and its information is read it can be seen that the certificate used is a the same as the one specified in *trust-anchors*. However, it is the public key which is different so the connection is not established.

```
subject: rdnSequence (0)
▼ rdnSequence: 6 items (id-at-commonName=SDM-WEB,id-at-organizational
  ► RDNSequence item: 1 item (id-at-countryName=ES)
  ► RDNSequence item: 1 item (id-at-stateOrProvinceName=Madrid)
  ► RDNSequence item: 1 item (id-at-localityName=Madrid)
  ► RDNSequence item: 1 item (id-at-organizationName=UC3M)
  ► RDNSequence item: 1 item (id-at-organizationalUnitName=UC3M)
  ► RDNSequence item: 1 item (id-at-commonName=SDM-WEB)
```

Figure 34: Rogue certificate

Furthermore, if the alert message capture in Wireshark is open it can be read that the alert is describing the situation where the certificate is unknown.

```
Transport Layer Security
▼ TLSv1.2 Record Layer: Alert (Level: Fatal, Description: Certificate Unknown)
  Content Type: Alert (21)
  Version: TLS 1.2 (0x0303)
  Length: 2
  ▼ Alert Message
    Level: Fatal (2)
    Description: Certificate Unknown (46)
```

Figure 35: Wireshark unknown certificate alert

4 Conclusion

This practice has definitely provided a different vision about software and applications in Android.

It has been the perfect example about how an application can be perceived as totally equal to its users but the security functionalities that it implements may vary a lot.

This is both good and bad, since an improvement in security is always positive for data privacy, it has also demonstrated that if the internal code related to security is modified by an attacker, is quite difficult to a normal user realise about that.

According to the practice development, it has been tedious as some parts of the provided code and instructions have been deprecated or included errors which have importantly delayed the project development. It has provoked that a huge part of the time dedicated to this practice has been employed in research tasks, which have enhanced our knowledge about different properties and functionalities related to Android development and security.

Furthermore, due to the fact that for some sections of the assignment there were little resources available on the Internet, these sections of assignment were developed by try and fail. Consequently, only after the finish of them a general understanding of the project was gained.

Summing up, this practice has been a good final point which made us realise about the importance of security in Android application, how it is supposed to be provided and why everyone should be concerned about its value.