Bipul Karki

CMPE 277

Smartphone Application Development, Section 1

Spring 2014

Extra Credit - Assignment
App Report,
Certificate pinning

INSTRUCTOR: Prof. Chandrasekar Vuppalapati

Certificate Pinning:

The "CertificatePinning" app demonstrates how to pin certificates to a default Apache HTTP client that is shipped with Android.

By default, when an app makes an SSL connection to a server, it checks if the server's certificate has a verifiable chain of trust back to a trusted (root) certificate and it matches the requested hostname. Relying on matching certificates between the Android device's trust store and the remote server opens up a security hole for "man-in-the-middle" attacks. Certificate pinning can be used to solve the MITM attacks problem. It means hard-coding the certificate known to be used by the server in the mobile application. The app can then ignore the device's trust store and rely on its own, and allow only SSL connections to hosts signed with certificates stored inside the application [1].

This also gives a possibility of trusting a host with a self-signed certificate without the need to install additional certificates on the device.

Advantages:

- Increased security - with pinned SSL certificates, the app is independent of the device's trust store. Compromising the hard coded trust store in the app is not so easy - the app would need to be decompiled, changed and then recompiled again - and it can't be signed using the same Android keystore that the original developer of the app used.
- Reduced costs - SSL certificate pinning gives you the possibility to use a self-signed certificate that can be trusted. For example, you're developing an app that uses your own API server. You can reduce the costs by using a self-signed certificate on your server (and pinning that certificate in your app) instead of paying for a certificate. Although a bit convoluted, this way, you've actually improved security and saved yourself some money.

Disadvantage:

Less flexibility - when you do SSL certificate pinning, changing the SSL certificate is not that easy. For every SSL certificate change, you have to make an update to the app, push it to Google Play and hope the users will install it.

How to implement Certificate Pinning on Android?

The purpose of this app is to simulate the certificate verifying process that can be done without creating your own website and certificate. So, instead this app obtains the certificate from https://www.google.com and stores it on its own keystore. The following section describes the process.

There are three steps in the process:

1. Obtain or create your own signed certificate for the desired host
2. Convert certificate to .bks format
3. Pin the certificate to an instance of standard HTTPS request.

1. Obtain a certificate:
In this project, the Google's SSL certificate was obtained using Firefox.

a. Click on the SSL certificate icon at the top (https://www.google.com).
b. Click `View Certificate`
c. Click on the `Details` Tab
d. Chose which certificate you want from the hierarchy [not circled in picture]
e. Click `Export`

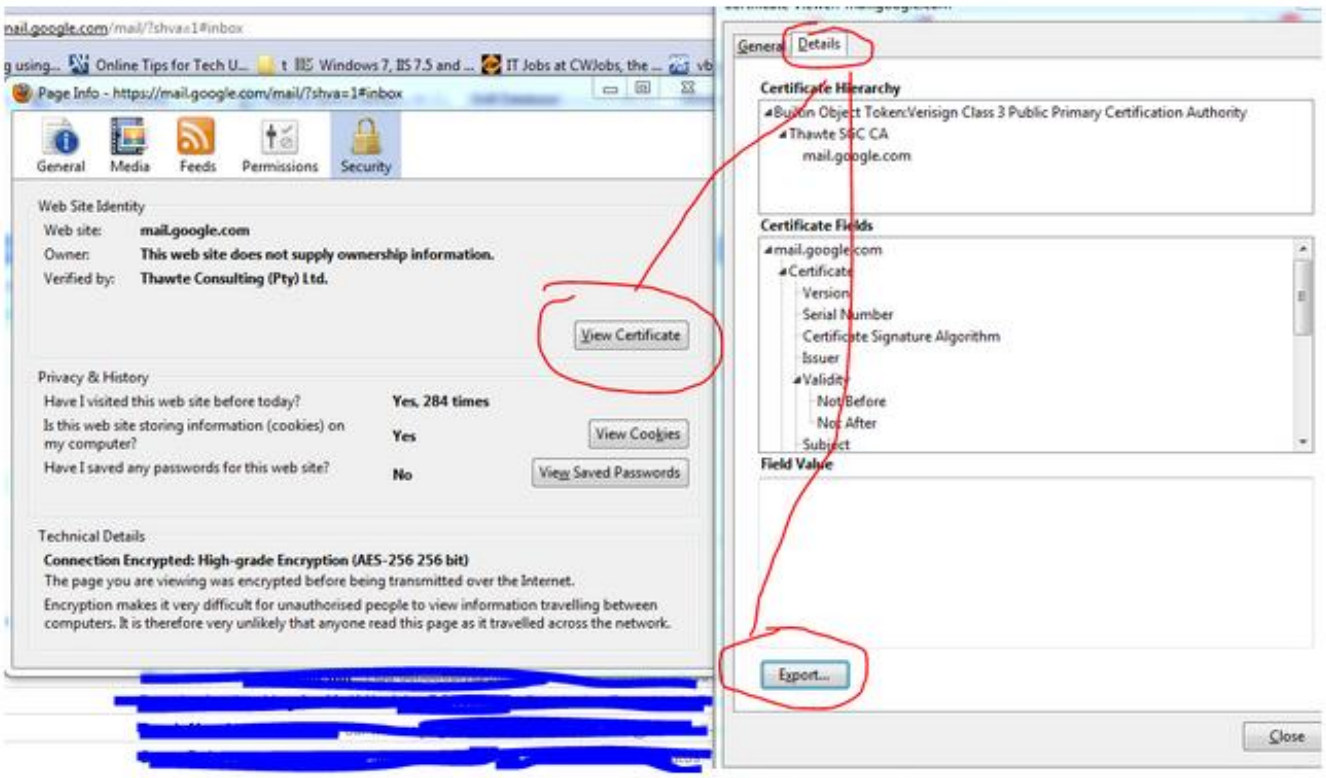


Fig. SSL certificate export in Firefox

2. Convert certificate to .bks format:

Firefox exports the SSL certificate in .crt format. In order to convert it to .bks format, Bouncy Castle is needed and the bouncycastle JAR can be downloaded from:
http://repo2.maven.org/maven2/org/bouncycastle/bcprov-ext-jdk15on/1.46/bcprov-ext-jdk15on-1.46.jar

The command to convert .crt to .bks format:

```
keytool -importcert -v -trustcacerts -file "mygoogle.crt" -alias ca -keystore
"keystore.bks" -provider org.bouncycastle.jce.provider.BouncyCastleProvider -
providerpath "bcprov-ext-jdk15on-1.46.jar" -storetype BKS -storepass testing
```

Use -file argument to specify .pem, .cert or .crt certificate file. Output keystore is specified using -keystore argument. Path to Bouncy Castle .jar must be provided with -providerpath argument. Finally password for the generated keystore is set with -storepass argument.

3. Pin the certificate to an instance of standard HTTPS request:

Put keystore.bks in the "assets" directory of the Android app and validate the server key against it. The following figure shows the method using a standard HTTPS request [2]:

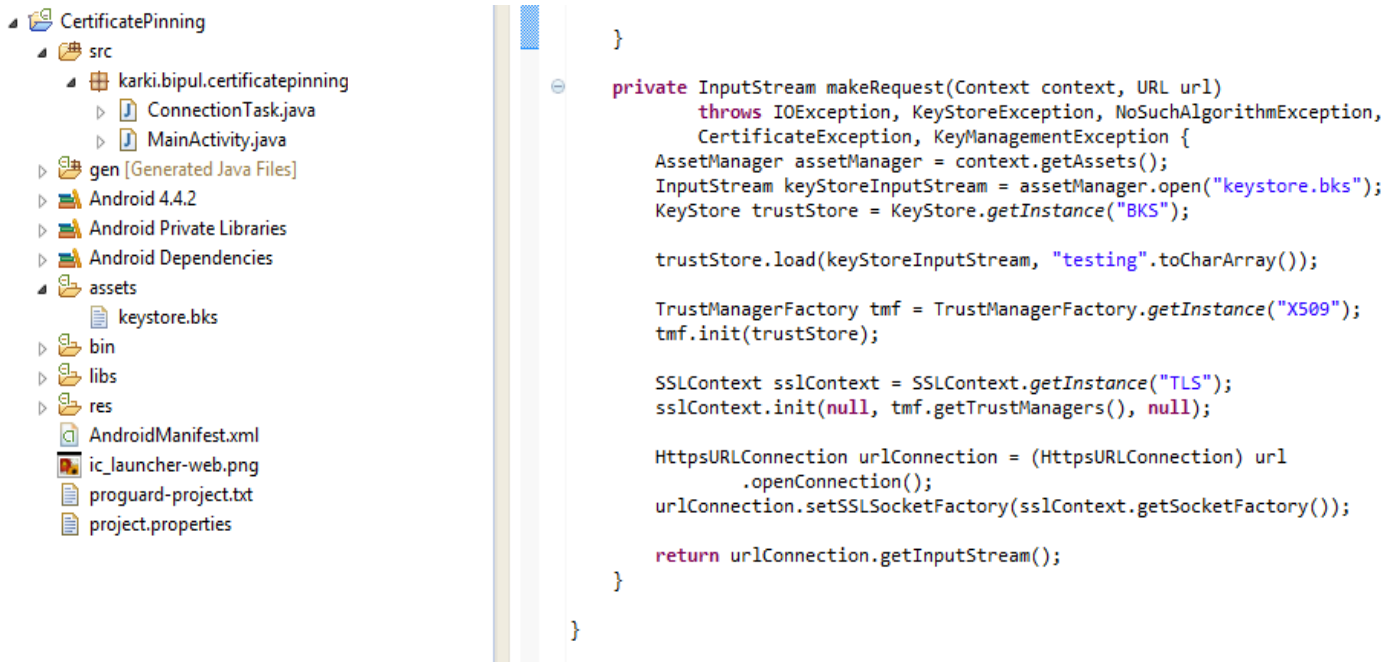


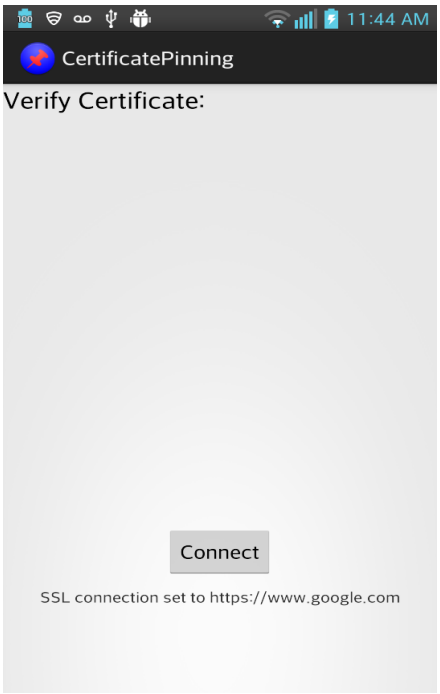
Fig. keystore.bks validation using standard HTTPS request

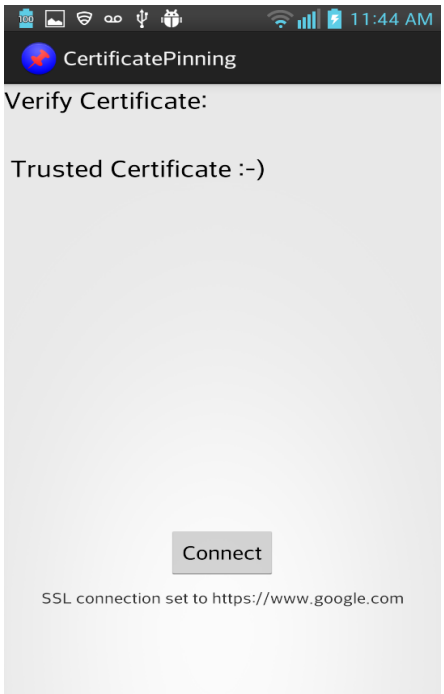
Running the App to check validation:

The default server location is https://www.google.com

```
@Override
protected Object doInBackground(Void... params) {

    Object result = null;

    try {
        URL url = new URL("https://www.google.com"); // trust only this
                                                     // site
        InputStream in = makeRequest(mContext, url);
        copyInputStreamToOutputStream(in, System.out);

    } catch (Exception ex) {

        // Log error
        Log.e("doInBackground", ex.toString());

        // Prepare return value
        result = (Object) ex;
    }

    return result;
}
```

When the app connects to above URL the SSL certificate is validated with local keystore certificate and the connection is established because the certificate now can be said from trusted source.





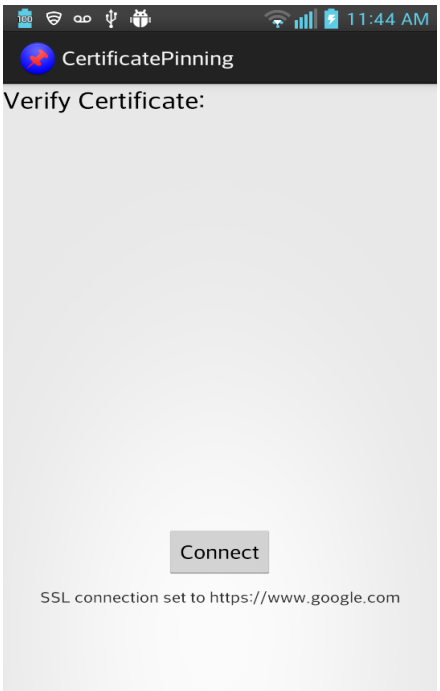
When the app tries to connect with other URLs, for example – https://www.facebook.com, the SSL certificate is validated with local keystore certificate which is for Google. The validation fails because the certificates do not match so the connection cannot be established.

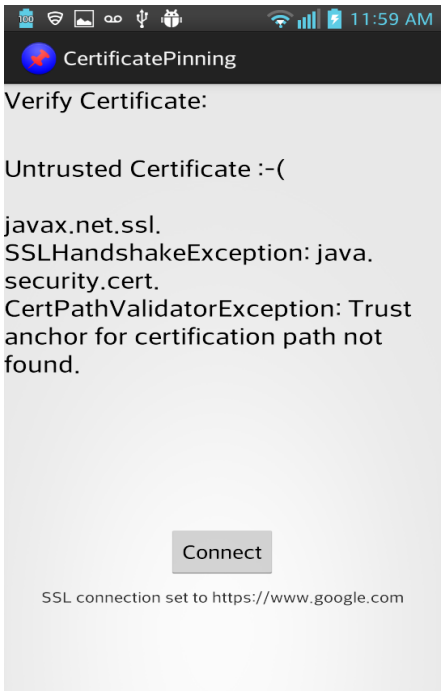
```
@Override
protected Object doInBackground(Void... params) {

    Object result = null;

    try {
        URL url = new URL("https://www.facebook.com"); // trust only this
                                                       // site
        InputStream in = makeRequest(mContext, url);
        copyInputStreamToOutputStream(in, System.out);

    } catch (Exception ex) {

        // Log error
        Log.e("doInBackground", ex.toString());

        // Prepare return value
        result = (Object) ex;
    }

    return result;
}
```

```
@Override
protected void onPostExecute(Object result) {
    // MainActivity.myText.setText("Test");
    if (result instanceof Exception) {
        MainActivity.myText.setText("Untrusted Certificate :-( \n\n"
                + result);
        return;
    }
    MainActivity.myText.setText(" Trusted Certificate :-)");
}
```

```
private InputStream makeRequest(Context context, URL url)
        throws IOException, KeyStoreException, NoSuchAlgorithmException,
        CertificateException, KeyManagementException {
    AssetManager assetManager = context.getAssets();
    InputStream keyStoreInputStream = assetManager.open("keystore.bks");
    KeyStore trustStore = KeyStore.getInstance("BKS");

    trustStore.load(keyStoreInputStream, "testing".toCharArray());

    TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
    tmf.init(trustStore);

    SSLContext sslContext = SSLContext.getInstance("TLS");
    sslContext.init(null, tmf.getTrustManagers(), null);

    HttpsURLConnection urlConnection = (HttpsURLConnection) url
            .openConnection();
    urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());

    return urlConnection.getInputStream();
}
```

The app checks Facebook's SSL certificate with its keystore and shows it is "Untrusted Certificate". The SSLHandshakeException occurs because there is a CA that isn't trusted by the system. It could also be because the certificate from a new CA that isn't yet trusted by Android or the app is running on an older version without the CA [3].





References

[1] I. Kust. "Securing mobile banking on Android with SSL certificate pinning." Internet: https://www.infinum.co/the-capsized-eight/articles/securing-mobile-banking-on-android-with-ssl-certificate-pinning, Mar. 12, 2014 [May 11, 2014].

[2] M. Marlinspike. "Your app shouldn't suffer SSL's problems." Internet: http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/, Dec. 5, 2011 [May 11, 2014].

[3] "Security with HTTPS and SSL." Internet: http://developer.android.com/training/articles/security-ssl.html, [May 11, 2014]