Secure Programming Lab 3: Solutions

Rui Li, Arthur Chan and David Aspinall

14th March 2017

1. Metadata and privacy

Checkpoint 1. Where did the metadata come from? Is any of the data concerning from a privacy perspective?

Mostly from the image and the camera app, but also some from the connection to the server. Much could be seen as concerning: the make and model of the camera could be used to identify users if only one has that device. The GPS coordinates are obviously concerning. The base64 encoded username and IP address are worrying on a site that is essentially pseudo-anonymous. The user-agent gives quite precise

Checkpoint 2. Pick a bit of metadata and describe how it could be used to attack, or deanonymise a user.

- **Image Model**: if only one person is using that device in your vicinity they’re probably the user.
- **Connecting-IP and User-Agent**: gives you a target to attack and a hint at what software they may be running.
- **Date Time and GPS**: this person must be taking Secure Programming!

Task: Some of the metadata has come from the EXIF tags embedded in the image. Modify the app to strip the EXIF tags.

Have a look at `android.media.ExifInterface.setAttribute()`.

Checkpoint 3. How accurate is the data? Where do the coordinates say the pictures were taken?

Not very. The app is set up to use the last known coarse location. This could change with time however. You’ll need to convert to decimal degrees and make the longitude negative since we’re west of Greenwich.

https://www.fcc.gov/media/radio/dms-decimal

You can then plot them on Google Maps. I usually turn up on Dalrymple Crescent, about a mile from the Informatics Forum where the picture was taken.

2. Transport Security

Checkpoint 4. Describe the size and format of the key. What is the certificate’s chain of trust? For what sites is the key valid?

- It’s an SSL certificate signed by a 2048 bit RSA key on a SHA256 Hash.
- The chain of trust goes:
  - DST Root CA X3 -> Let’s Encrypt Authority X3 -> infr11098.space

The certificate is only valid for `infr11098.space` (the **Common Name**).

Checkpoint 5. Many browsers have the certificate authority’s certificates built in. In Firefox they can be found in the **Certificates** tab under **Advanced** in `about:preferences`. Find the CA certificates used to sign `infr11098.space`.
Essentially the certificates identified in checkpoint 3.

**Task:** Modify the app to use the HTTPS protocol if the server address requires it. If an error occurs you should stop and report the message to the user.

Essentially just switch the HttpURLConnection for an HttpURLConnection. Some switching based on the url.getProtocol would be nice.

**Checkpoint 6.** Does the connection still work? If not why not?

Nope. The certificate is not valid for the ip address, it only valid for the domain name infr11908.space. That is not the domain embedded in the certificate, even though they both point to the same server.

### 3. Pinned Certificates

**Checkpoint 7.** Think about the two attacks, and the work required for an attacker. What certificates would the attacker need for their deception to work with HTTPS? How might the user notice they were being attacked?

- **Simple Approach:** relies on user not spotting the typo. The attacker would need a certificate for their own domain, which should be easily available. Some browsers may report that the domain is probably not the one you intended to visit.

- **MITM:** relies on you tricking the user into installing your fake certificate, or getting a CA to issue you with a certificate (which they should not do). Users may notice the certificate is different if they are weirdly observant. DigiNotar were caught issuing bad certificates for google.com in 2011. The National Informatics Center of India and the Indian Controller of Certifying Authorities were caught in 2014.

**Task:** Implement certificate pinning for the app and the server.

*Moxie Marlinspike* (Ex-head of security at Twitter, and Open Whisper Systems founder) has written a library to do this. The code is freely available and is worth reading for a guide to doing it correctly.

- [https://github.com/moxie0/AndroidPinning](https://github.com/moxie0/AndroidPinning)
package org.thoughtcrime.ssl.pinning;

import android.util.Log;

import java.security.KeyStoreException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;
import javax.net.ssl.X509TrustManager;

/**
 * A TrustManager implementation that enforces Certificate "pins."
 *
 * <p>
 * PinningTrustManager is layered on top of the system's default TrustManager,
 * such that the system continues to validate CA signatures for SSL connections
 * as usual. Additionally, however, PinningTrustManager will enforce certificate
 * constraints on the validated certificate chain. Specifically, it
 * will ensure that one of an arbitrary number of specified SubjectPublicKeyInfos
 * appears somewhere in the valid certificate chain.
 * </p>
 * <p>
 * <p>
* To use:
* <pre>
* TrustManager[] trustManagers = new TrustManager[1];
* trustManagers[0] = new PinningTrustManager(SystemKeyStore.getInstance(),
*     new String[] {"f30012bbc18c231ac1a44b788e410ce754182513",
*                     0});
* 
* SSLContext sslContext = SSLContext.getInstance("TLS");
* sslContext.init(null, trustManagers, null);
* 
* HttpsURLConnection urlConnection = (HttpsURLConnection) new URL("https://encrypted.google.com/").openConnection();
* urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());
* InputStream in = urlConnection.getInputStream();
* </pre>
* </p>
* @author Moxie Marlinspike
*/

public class PinningTrustManager implements X509TrustManager {

    private final TrustManager[] systemTrustManagers;
    private final SystemKeyStore systemKeyStore;
    private final long enforceUntilTimestampMillis;

    private final List<byte[]> pins = new LinkedList<byte[]>();
    private final Set<X509Certificate> cache = Collections.synchronizedSet(new HashSet<X509Certificate>);

    /**
     * Constructs a PinningTrustManager with a set of valid pins.
     *
     * @param keyStore A SystemKeyStore that validation will be based on.
     *
     * @param pins An array of encoded pins to match a seen certificate chain against. A pin is a hex-encoded hash of a X.509 certificate's
     * SubjectPublicKeyInfo. A pin can be generated using the provided pin.py script: python ./tools/pin.py certificate_file.pem
     *
     * @param enforceUntilTimestampMillis A timestamp (in milliseconds) when pins will stop being enforced. Normal non-pinned certificate validation
     * will continue. Set this to some period after your build date, or to 0 to enforce pins forever.
     */
    public PinningTrustManager(SystemKeyStore keyStore, String[] pins, long enforceUntilTimestampMillis) {
        this.systemTrustManagers = initializeSystemTrustManagers(keyStore);
        this.systemKeyStore = keyStore;
        this.enforceUntilTimestampMillis = enforceUntilTimestampMillis;

        for (String pin : pins) {
            this.pins.add(hexStringToByteArray(pin));
        }
    }
private TrustManager[] initializeSystemTrustManagers(SystemKeyStore keyStore) {
    try {
        final TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
        tmf.init(keyStore.trustStore);

        return tmf.getTrustManagers();
    } catch (NoSuchAlgorithmException nsae) {
        throw new AssertionError(nsae);
    } catch (KeyStoreException e) {
        throw new AssertionError(e);
    }
}

private boolean isValidPin(X509Certificate certificate) throws CertificateException {
    try {
        final MessageDigest digest = MessageDigest.getInstance("SHA1");
        final byte[] spki = certificate.getPublicKey().getEncoded();
        final byte[] pin = digest.digest(spki);

        for (byte[] validPin : this.pins) {
            if (Arrays.equals(validPin, pin)) {
                return true;
            }
        }

        return false;
    } catch (NoSuchAlgorithmException nsae) {
        throw new CertificateException(nsae);
    }
}

private void checkSystemTrust(X509Certificate[] chain, String authType)
    throws CertificateException {
    for (TrustManager systemTrustManager : systemTrustManagers) {
        ((X509TrustManager) systemTrustManager).checkServerTrusted(chain, authType);
    }
}

private void checkPinTrust(X509Certificate[] chain)
    throws CertificateException {
    if (enforceUntilTimestampMillis != 0 &&
        System.currentTimeMillis() > enforceUntilTimestampMillis)
    {
        Log.w("PinningTrustManager", "Certificate pins are stale, falling back to system trust.");
        return;
    }

    final X509Certificate[] cleanChain = CertificateChainCleaner.getCleanChain(chain, systemKeyStore);

    for (X509Certificate certificate : cleanChain) {
        if (isValidPin(certificate)) {

public void checkClientTrusted(X509Certificate[] chain, String authType) 
    throws CertificateException {
    throw new CertificateException("Client certificates not supported!");
}

public void checkServerTrusted(X509Certificate[] chain, String authType) 
    throws CertificateException {
    if (cache.contains(chain[0])) {
        return;
    }

    // Note: We do this so that we'll never be doing worse than the default
    // system validation. It's duplicate work, however, and can be factored
    // out if we make the verification below more complete.
    checkSystemTrust(chain, authType);
    checkPinTrust(chain);
    cache.add(chain[0]);
}

public X509Certificate[] getAcceptedIssuers() {
    return null;
}

private byte[] hexStringToByteArray(String s) {
    final int len = s.length();
    final byte[] data = new byte[len / 2];

    for (int i = 0; i < len; i += 2) {
        data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4) +
            Character.digit(s.charAt(i + 1), 16));
    }

    return data;
}

public void clearCache() {
    cache.clear();
}

PinningSSLSocketFactory
/**
package org.thoughtcrime.ssl.pinning;

import android.content.Context;
import org.apache.http.conn.ssl.SSLSocketFactory;
import org.apache.http.conn.ssl.X509HostnameVerifier;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.TrustManager;
import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.security.KeyManagementException;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;

/**
 * A standard Apache SSL Socket Factory that uses an pinning trust manager.
 *<p>
 * To use:
 *</p>
 *<pre>
 * String[] pins = new String[] {"40c5401d6f8cbaf08b00edefb1ee87d005b3b9cd"};
 * SchemeRegistry schemeRegistry = new SchemeRegistry();
 * schemeRegistry.register(new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));
 * schemeRegistry.register(new Scheme("https", new PinningSSLSocketFactory(getContext(),pins, 0), 443));
 *
 * HttpParams httpParams = new BasicHttpParams();
 * ClientConnectionManager connectionManager = new ThreadSafeClientConnManager(httpParams, schemeRegistry);  
 * DefaultHttpClient httpclient = new DefaultHttpClient(connectionManager, httpParams);
 */
public class PinningSSLSocketFactory extends SSLSocketFactory {

    private final javax.net.ssl.SSLSocketFactory pinningSocketFactory;

    /**
     * Constructs a PinningSSLSocketFactory with a set of valid pins.
     * @param pins An array of encoded pins to match a seen certificate chain against. A pin is a hex-encoded hash of a X.509 certificate's SubjectPublicKeyInfo. A pin can be generated using the provided pin.py script: python ./tools/pin.py certificate_file.pem
     * @param enforceUntilTimestampMillis A timestamp (in milliseconds) when pins will stop being enforced. Normal non-pinned certificate validation will continue. Set this to some period after your build date, or to 0 to enforce pins forever.
     */

    public PinningSSLSocketFactory(Context context, String[] pins, long enforceUntilTimestampMillis) throws UnrecoverableKeyException, KeyManagementException, NoSuchAlgorithmException, KeyStoreException {
        super(null);
        final SystemKeyStore keyStore = SystemKeyStore.getInstance(context);
        final SSLContext pinningSslContext = SSLContext.getInstance(TLS);
        final TrustManager[] pinningTrustManagers = initializePinningTrustManagers(keyStore, pins, enforceUntilTimestampMillis);

        pinningSslContext.init(null, pinningTrustManagers, null);
        this.pinningSocketFactory = pinningSslContext.getSocketFactory();
    }

    @Override
    public Socket createSocket() throws IOException {
        return pinningSocketFactory.createSocket();
    }

    @Override
    public Socket connectSocket(final Socket sock, final String host, final int port, final InetAddress localAddress, int localPort, final HttpParams params) throws IOException {
        final SSLSocket sslSock = (SSLSocket) ((sock != null) ? sock : createSocket());

        if ((localAddress != null) || (localPort > 0)) {
            if (localPort < 0) {
                // Additional code here
            } else {
                // Additional code here
            }
        } else {
            // Additional code here
        }
    }
}
localPort = 0;
}
sslSock.bind(new InetSocketAddress(localAddress, localPort));

final int connTimeout = HttpConnectionParams.getConnectionTimeout(params);
final int soTimeout = HttpConnectionParams.getSoTimeout(params);

final InetSocketAddress remoteAddress = new InetSocketAddress(host, port);
sslSock.connect(remoteAddress, connTimeout);
sslSock.setSoTimeout(soTimeout);

try {
    SSLSocketFactory.STRICT_HOSTNAME_VERIFIER.verify(host, sslSock);
} catch (IOException iox) {
    try {
        sslSock.close();
    } catch (Exception ignored) {
    }
    throw iox;
}

return sslSock;

@Override
public Socket createSocket(final Socket socket, final String host,
    int port, final boolean autoClose)
    throws IOException {
    if (port == -1) {
        port = 443;
    }

    final SSLSocket sslSocket = (SSLSocket) pinningSocketFactory.createSocket(socket, host, port, autoClose);
    SSLSocketFactory.STRICT_HOSTNAME_VERIFIER.verify(host, sslSocket);
    return sslSocket;
}

@Override
public void setHostnameVerifier(X509HostnameVerifier hostnameVerifier) {
    throw new IllegalArgumentException("Only strict hostname verification (default) " +
            "is supported!");
}

@Override
public X509HostnameVerifier getHostnameVerifier() {
    return SSLSocketFactory.STRICT_HOSTNAME_VERIFIER;
}

private TrustManager[] initializePinningTrustManagers(SystemKeyStore keyStore,
String[] pins,
long enforceUntilTimestampMillis)
{
    final TrustManager[] trustManagers = new TrustManager[1];
    trustManagers[0] = new PinningTrustManager(keyStore, pins, enforceUntilTimestampMillis);

    return trustManagers;
}
}